Technical Report 02-2008

XPERSIF: A Software Integration Framework & Architecture for Robotic Learning by Experimentation.

Iman Awaad
Beatriz Leon

Februay 2008

Publisher: Dean Prof. Dr. Kurt Ulrich Witt

University of Applied Sciences Bonn-Rhein-Sieg,
Department of Computer Science

Sankt Augustin, Germany

Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

ISSSN 1869-5272

ABSTRACT

The integration of independently-developed applications into an efficient system, particularly in a distributed setting, is the core issue addressed in this work.

Cooperation between researchers across various field boundaries in order to solve complex problems has become commonplace. Due to the multidisciplinary nature of such efforts, individual applications are developed independent of the integration process. The integration of individual applications into a fully-functioning architecture is a complex and multifaceted task. This thesis extends a component-based architecture, previously developed by the authors, to allow the integration of various software applications which are deployed in a distributed setting.

The test bed for the framework is the EU project XPERO, the goal of which is robot learning by experimentation. The task at hand is the integration of the required applications, such as planning of experiments, perception of parametrized features, robot motion control and knowledge-based learning, into a coherent cognitive architecture. This allows a mobile robot to use the methods involved in experimentation in order to learn about its environment.

To meet the challenge of developing this architecture within a distributed, heterogeneous environment, the authors specified, defined, developed, implemented and tested a component-based architecture called XPERSIF. The architecture comprises loosely-coupled, autonomous components that offer services through their well-defined interfaces and form a service-oriented architecture. The Ice middleware is used in the communication layer. Its deployment facilitates the necessary refactoring of concepts. One fully specified and detailed use case is the successful integration of the XPERSim simulator which constitutes one of the kernel components of XPERO.

The results of this work demonstrate that the proposed architecture is robust and flexible, and can be successfully scaled to allow the complete integration of the necessary applications, thus enabling robot learning by experimentation. The design supports composability, thus allowing components to be grouped together in order to provide an aggregate service. Distributed simulation enabled real time tele-observation of the simulated experiment. Results show that incorporating the XPERSim simulator has substantially enhanced the speed of research and the information flow within the cognitive learning loop.

ACKNOWLEDGMENTS

We would like to express our sincere gratitude to our supervisors: Karl-Heinz Sylla of the Institute for Autonomous Intelligent Systems, and to Professor Paul Plöger and Ronny Hartanto of the University of Applied Sciences Bonn-Rhein-Sieg for their effective guidance and tremendous support. Their patience and generosity with their time and effort have made all the difference.

We would like to especially thank Professor Erwin Prassler for providing us with the opportunity to participate in the XPERO project. Many thanks to Professor Gerhard Kraetzschmar and to our former supervisor Keyan Ghazi-Zahedi for their dedicated commitment and lasting contribution to our characters, skills and academic and professional developments. We also gratefully acknowledge Professor Norbert Jung's mentorship in helping us find our way. In addition, we would like thank our colleagues both within the XPERO project and outside it for the invaluable discussions and their helpful feedback. Our gratitude also goes to our classmates who started their journey with the master's programme with us and who have been like our extended family, always lending their support and their time and helping to keep us going. Most notably, Mr. Varun Ghai who was always near, even when he was far.

Last but not least, we ingratiate ourselves to our families for their constant encouragement, their unreserved support, their patience, their understanding and the sacrifices they have made for us throughout our studies. Thank you.

The work described in this article has been [partially] funded by the European Commission's Sixth Framework Programme under contract no. 029427 as part of the Specific Targeted Research Project XPERO ("Robotic Learning by Experimentation").

CONTENTS

Chapter 1

INTRODUCTION

This thesis aims to elaborate on a software integration framework and architecture for robotic learning by experimentation. This chapter provides an introduction to the project, its software requirements and the challenges that must be dealt with, revealing the motivation for the thesis and outlining the chosen approach before presenting an overview of the results.

1.1 Motivation

Computerized solutions have developed from a simple algorithm, to programs that might contain more than one of these, to groups of programs forming an application. Nowadays these solutions might encompass numerous applications running on a number of machines. More often, these applications are developed independently and must be integrated into an architecture. Along with these developments, the complexity in the task of designing and abstracting (or architecting) these architectures has also grown. Principles that guide the structuring of such distributed applications are necessary, as is the use of technology which facilitates their development.

This thesis employs the principles of service-oriented architecture and technology such as middleware to integrate heterogeneous applications into a cognitive architecture for the XPERO project whose goal is robotic learning by experimentation.

The learner is an embodied agent, not an algorithm running on a computer as it has been for decades, which is able to apply the scientific method (as seen in figure 1.1) used in experimentation from start to finish. This is a fundamental step forward for learning and robotics as it paves the way for open-ended learning. The cost of this step forward is that researchers must deal with both the complexity involved in the learning process in addition to that involved in operating a robotic system.

Within the context of the project, a robot is expected to make observations of its environment, and predict how the environment will behave (the default state of the robot). When an observation does not match what has been predicted, the robot – motivated by surprise for example – is expected to devise an experiment that should enable it to explain this discrepancy in observations and perhaps update its prediction model. This experimentation process may also be triggered by internal motivational cues

such as curiosity or hunger. It will then execute the experiment(s) and once again use the observations of the results to revise the hypotheses or generate new ones altogether.

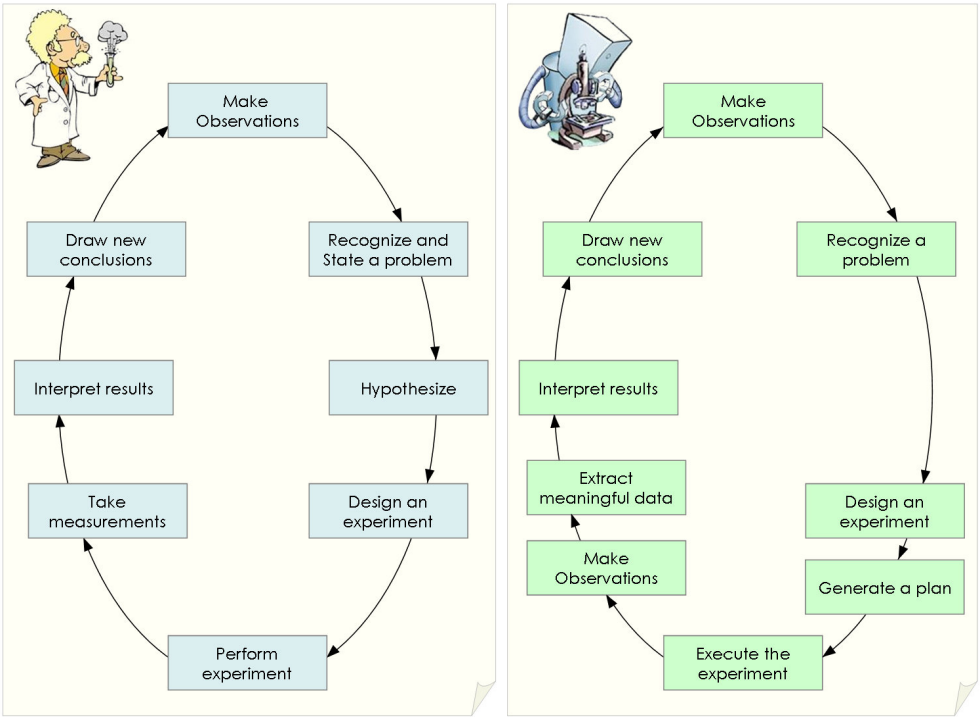

Figure 1.1: The scientific method – for the human researcher on the left and the robotic experimenter on the right – depicted as an experimental loop.

Here the experimental loop in figure 1.1 is to be understood as a cyclic sequence of performing an experiment in the robot's world, tracing a well defined set of either raw sensor data or pre-processed feature-based percepts, which are then fed to respective machine learning tools. The results of this processing step is in turn used to devise new experiments to be performed by the robot to gain more insights about its world. To enable this loop, various software and hardware systems are necessary. The challenge lies not only in their individual implementation but in their integration into an efficient and robust single system that allows the robot to close the loop and learn by experimentation. Due to the high computational demands of these systems, they must be distributed across different machines. Moreover, the partners that implement these systems for the project are distributed geographically across Europe and each uses a different platform. These circumstances add an additional layer of complexity to the task of integration.

1.2 Previous work

We designed a component-based software integration framework for XPERO called XPERSIF [4] and implemented the first integration of an XPERO application, namely the simulator called XPERSim [3], into the framework. The work allowed tele-operation and tele-observation of a simulated robot experiment and laid the groundwork for future integration of additional XPERO applications. An introduction to the XPERSIF framework and to the XPERSim simulator can be found in sections 4.1 and 2.5.2 respectively.

1.3 Problem statement and thesis goals

The goal of this thesis is to further develop, implement, and evaluate the software integration framework and architecture: XPERSIF for robot learning by experimentation. This allows the further enhancement of the speed of the experimental loop through the integration of more of the participating partners' applications while focusing on achieving the specified scenarios which are used within the XPERO project.

Specifically, this work is accomplished by first extending the existing XPERSIF architecture from its current state to allow it to scale in an efficient manner in anticipation of the complete integration of XPERO applications. A series of use cases is defined as part of the requirements specification for the software system. An analysis of these requirements is used to design both the individual interfaces for the components which allow data to flow between them and services to be accessed as well as the organizational layer which manages them.

The implementation focus of the work is aimed specifically at providing base applications and interfaces for the partners' systems. In addition, an efficient solution to distributing the simulation to numerous users will be implemented.

This thesis will argue that the design of the framework not only allows for efficiently closing the loop thus enabling robot experimentation but that incorporating a simulator has enhanced the speed of research and the information flow within the loop. Moreover, this thesis will demonstrate a robust and flexible architecture within the setting of robot learning by experimentation that is able to successfully and efficiently detect faults and handle errors in a structured manner.

The complete integration of the applications into a workbench for robot learning by experimentation will not be completed as part of this thesis, but is planned for subsequent efforts.

1.4 The approach

A component based software engineering (CBSE) approach has been used to encapsulate the functionalities of the software and hardware systems mentioned above into components. These components have been loosely coupled to form a Service-Oriented Architecture (SOA). An introduction to CBSE and SOA in addition to the component model used in this work can be found in Chapter 2. The Ice middleware has been used to allow distribution of the framework and to facilitate the communication between the various components. In addition, several services provided by the middleware, such as grid computing, have been used to address a number of issues related to distributed application development.

As mentioned above, the process of specifying requirements, analyzing them, designing a system which satisfies these requirements, implementing the system and evaluating and refining it is followed. This is often referred to as the waterfall method in software engineering terms. These processes result in use cases, the architectural design and the interface definitions for the components before finally culminating in the actual application bases and their integration and evaluation.

The approach used to distribute the simulation is based on techniques adopted from the field of multi-player games in which many users share the same rendered experience in a quasi-real time manner.

1.5 Major results

A collection of use cases were modelled in UML formalizing the system's functional requirements, thus providing a valuable repository for developing and validating the system. The resulting software architecture easily and efficiently allows the integration of components (both software and hardware) which run on heterogeneous platforms and languages. The use of CBSE allows the software architecture to maximize concurrency in the application development process of the various research groups. The adoption of the SOA approach in the design of the framework has produced a system which is highly flexible and maintainable. The framework is data-centric with communication of the data playing a significant role in the design. The simplicity of the communication patterns contributes to the efficiency and flexibility of the framework. The data itself which is exchanged between components is abstracted in such a way as to maintain interfaces which are as simple as possible. Additionally, the solution for the tele-observation of experiments is a significant contribution to the framework as a whole.

1.6 Thesis overview

Chapter 2 will discuss and compare related work and provide the necessary background information on the chosen approaches. Specifically, an overview of service-oriented architecture is presented along with sections on middleware, simulation and communication patterns. Chapter 3 will present the requirements for the software framework and a series of use cases that formalize the specifications. Chapter 4 will detail the XPERSIF framework and system architecture, presenting the component model which forms the basis for all components within the loop, in addition to the data and control flow between them. It will also analyze the various communication patterns as used within the architecture. Chapter 5 details the system's implementation. Chapter 6 will convey the specifics of the evaluation process and present the results. Chapter 7 will synthesize the experimental data, draw conclusions, and discuss open issues and lessons learned. Appendix A presents the typographical conventions used within this work. Appendix B contains the compilation of Slice interface definitions and Appendix C provides instructions on setting up an application on an Ice Grid node. Appendix D discusses the details of the motion controller. Appendix E specifies the authorship of the various chapters.

Chapter 2

BASICS

This chapter provides basic knowledge on several topics which are significant to this thesis. First, the topic of service-oriented architecture, which is a school of thought from which this work draws inspiration, is introduced. A brief history of the approach is also outlined. Next, we address the topic of middleware which facilitates the development of distributed applications before discussing related work. We conclude with a section on simulation which has been critical to the research done within the XPERO project and is an implementation focus of this work.

2.1 Service-oriented architecture

As mentioned in section 1.1, the complexity of architecting has grown along with the developments in computing. Architectures such as 'client-server', 'peer-to-peer', 'multi-tier' and 'blackboard' emerged as templates – blueprints for the application architects to be used in similar manner to those used by building architects. More than one architecture may be used within an organization and, in such cases, an enterprise architecture is then defined to specify how these architectures co-exist and perhaps even integrate. An enterprise architecture is to an organization what an urban plan is to a city [11].

The scope of service-oriented architectures (SOAs) includes both enterprise and application architectures. '*Service-orientation*' is a design paradigm for architecting a distributed architecture which centers around loosely-coupled autonomous components and groups of components providing services in order to carry out a given task. The principles of this paradigm allow the separation of the business and application logic domains of an enterprise. They guide their structuring into the layers associated with a SOA model (figure 2.1). Additionally, they provide tenants for the granularity and other characteristics towards which individual services should strive. SOAs are basically a collection of services. A service is well-defined and self-contained, and does not depend on the context or state of other services [11]. These services can be registered with a central registry which allows service requesters to discover them. This is the essence – the abstract idea of a SOA.

How a SOA is implemented can vary. An implementation approach based on that of traditional distributed architectures, or alternatively one utilizing 'web services' (WS)

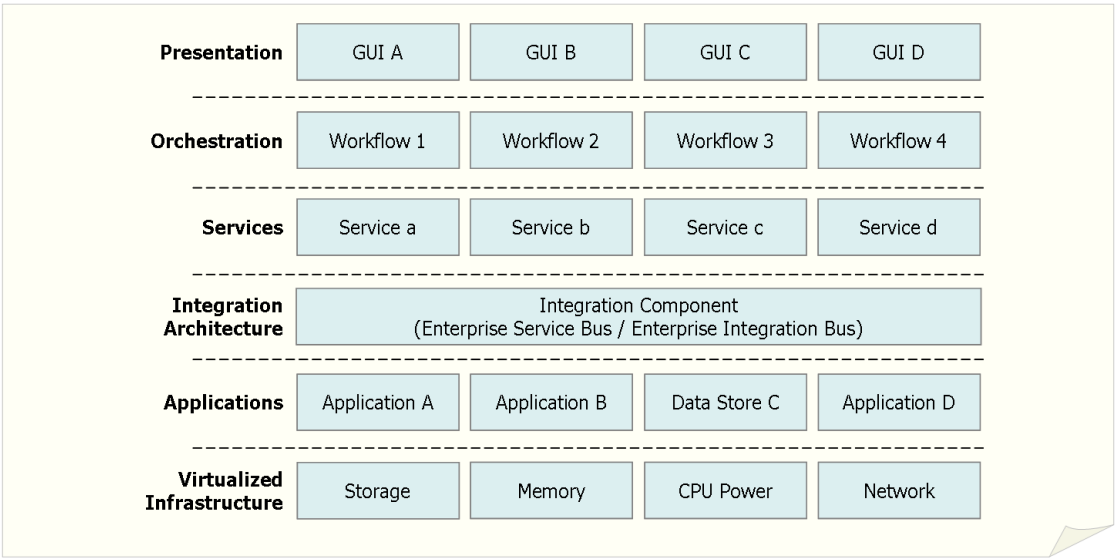

Figure 2.1: The layers of a SOA Model (reproduced from [32]).The XPERSIF framework provides the integration architecture for the SOA (the central layer of the figure).

which takes advantage of the Internet and its proprietary-free communications network may be adopted. Such a SOA might use document-style messages which are encoded in protocols such as the Simple Object Access Protocol (SOAP) to transport, process and route the payload which itself is represented in the Xtensible Markup Language (XML).

In this work, the traditional distributed architecture is used in the implementation of a SOA. The composability of SOA is achieved through the use of component-based software engineering techniques (CBSE).

CBSE adopts the doctrine of 'divide and conquer' by breaking down a system into functional or logical components. These components (and the services they provide) are accessed through their interfaces which are carefully specified. This makes it possible for the actual implementation details (algorithms, etc) of a component to be hidden behind its interface, thus enabling plug and play functionality through shared interfaces for components using varying implementations to provide the same service. These modular and reusable components can be coupled and layered to build larger systems which are robust and easy to maintain. Their development and testing is simpler, as each component is developed independently from other components (while sharing a common component model). This characteristic means that an expert in each component's functionality can develop the component independently based on a component model without needing to possess expertise in all issues relating to the system as a whole. This makes the development time shorter (as work can proceed in parallel) than that needed

for non-CB systems. The component model used for this work is defined in section 4.1.

Components can be placed on various computers – dedicated application servers – to form a distributed architecture. Traditionally, Remote Procedure Calls (RPCs) are used for communication between components within such a distributed architecture. This is a point of difference between SOAs based on these distributed architectures and those based on web services, as the communication between services in the latter case is accomplished through document-style messages using protocols such as SOAP, which are as self-sufficient as possible (containing policy rules, meta-information and processing instructions for example). This tends to result in larger messages that are sent and received less frequently in comparison to RPC communication as used in traditional distributed architectures.

Some of the main principles behind SOAs such as modularization and reuse of services can be traced back to the object-oriented (OO) approach. While OO strives to follow such principles at the function and object levels, SOAs strive to do this at the service level allowing services to be reused and aggregated. SOAs also stress inter-operability (the ability for a service to run on any platform). Following is a list of the basic principles of SOA (as stated in [11]) which the design of the XPERSIF framework and architecture has strived to uphold to varying degrees:

- service loose coupling (which enables autonomy and supports composability)
- service autonomy (which increases the quality of composability and supports reusability)
- service composability (which is supported by the rest of the principles)
- service reuse (which enables composability)
- service contracts (which form the basis of discoverability and provide descriptions in support of abstraction)
- service abstraction (which packages services in support of reusability)
- service statelessness (which increases the quality of service composability)

2.2 Communication patterns

As mentioned in the previous section, communication plays a large role in an architecture. The design decisions, such as how often and in which representation communication is carried out, heavily impact the functionality of the architecture.

Software patterns represent recurring solutions to software development problems within a particular context. They facilitate reuse by providing templates of 'traditional wisdom' solutions to recurring design problems. Communication patterns are thus common approaches which are used in various settings for communication. This section provides an overview of a number of common communication patterns which are used in XPERSIF. Section 4.3 details the application of these patterns within XPERSIF.

Almost all communication is based on the *request-response* pattern where one party makes a request and the other party provides a response. This bidirectional transmission can be implemented as synchronous or asynchronous communication.

The *fire-and-forget* pattern involves the uni-directional transmission of a message from a single source to one or more destinations. A variety of patterns based on this simple pattern include multi-cast and broadcast patterns where a message is sent to numerous destinations. Streaming is a form of broadcasting (often used for transmitting multimedia) where the message is transmitted over a network in a real time manner and where quality of service (QoS) is an issue to be dealt with.

The *publish-subscribe* pattern is an asynchronous communication pattern in which a party (the publisher) transmits a message relating to a specific topic to one or more parties (subscribers) who have asked to receive that specific topic's messages. This pattern is often used in implementing the observer behavioural pattern which allows automatic notifications of an object's state change to be received by its dependents. Figure 2.2 shows the publish-subscribe pattern implemented using both the request-response pattern and the fire-and-forget patterns.

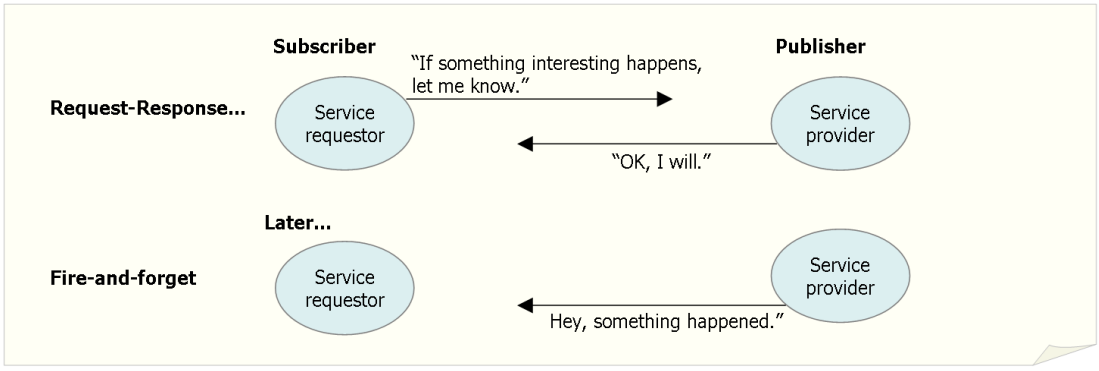

Figure 2.2: The publish-subscribe pattern implementation using the request-response and fire-and-forget patterns (based on the diagram in [11]).

2.3 Middleware

This section will provide basic knowledge about middleware and its use in developing distributed applications. It also provides a brief comparison of various middleware. The middleware requirements of the XPERO project are then specified before providing useful information on the chosen middleware: Ice and the various Ice services which are currently being used.

2.3.1 Introduction to middleware

Object-oriented middleware makes it possible for application developers to write applications that will run in a distributed manner without having to worry about the details of network communications (e.g. marshalling – the process of serializing a complex data structure, such as a sequence or a dictionary, for transmission on the wire [20] – and unmarshalling, etc).

Middleware is computer software that connects software components or applications. It is used most often to support complex, distributed applications. It includes web servers, application servers, content management systems, and similar tools that support application development and delivery [33]. Some examples of middleware are CORBA [18] from OMG, Ice [20] from ZeroC, DCOM [17] and .Net [10] from Microsoft, and Miro [30].

The first step in writing a distributed application is to specify interfaces. An interface is a collection of named operations that are supported by an object. Clients issue requests by invoking operations [20]. These interfaces are specified using a language. In CORBA, for example, this language is known as the Interface Definition Language, or IDL. In Ice, a language called Slice is used. The differences between these two languages are few but they reflect basic differences in the two middlewares.

These differences include functionality which is provided by Slice and not by IDL, for example, the use of dictionaries and the inheritance of exceptions (both of which have been reliably and efficiently used in the work presented here). Similarly, a number of features which IDL offers are not present in Slice. These mostly relate to CORBA's efforts to provide several ways to achieve the same task. A major drawback is the steep learning curve for application developers needed to use these different methods. This also affects the code size of the CORBA middleware itself. A related issue is the use of keywords within the interface definition language itself. For example, both CORBA and Ice offer one-way invocations (used in this work for distributing the simulation in section 5.6), the *oneway* keyword is part of the CORBA IDL (even though it does not relate to an interface – it is not part of the contract which is the interface between the client and

the server). While the service is offered in Ice, the keyword is not found within Slice itself. These differences between the two interface specification languages reflect the differences between the two middlewares, most notably the difference in complexity and size both of which greatly impact the ease of use and, as such, development time. For more details the reader is referred to [20].

Whichever language the interfaces are specified in, they are then compiled to generate files that contain the information and code necessary for the middleware to perform the networking tasks in order to allow the user to call the operations within the interfaces. The user then writes a client and a server for the application and deploys it.

While alternate object-oriented middleware platforms may provide their own solution to the problem of developing distributed applications, each comes with its own drawbacks. DCOM and .NET, for example, may be a solution for Microsoft applications, but do not address the problem when the network contains heterogeneous operating systems (which is the case in the XPERO project). In addition, they are very complex, requiring much time for developers to learn to use them. This drawback is also shared by CORBA. While CORBA is available from various vendors, it is difficult to find standards-compliant versions that perform well [20]. SOAP and web services provide yet more solutions, however these too suffer from serious drawbacks mostly stemming from issues relating to the lack of standardization, the high level of performance needed (both in terms of CPU and network bandwidth) and even security issues. Additionally, the need to use proprietary development platforms robs application developers of choosing other alternatives.

2.3.2 XPERO's middleware requirements

Various criteria for the evaluation of existing middleware solutions were used to narrow the search for a middleware platform that satisfies the needs of the XPERO project. One criterion was performance as determined in terms of speed. Yet another was the range of services which are offered. It was important that the middleware not demand extensive time and effort to learn and use. Cost and the availability of the source code were also used as criteria, as was the availability of the middleware through more than one vendor. In considering the above criteria, Ice emerged as the best choice of middleware for the project's requirements.

2.3.3 An introduction to the Ice middleware

This section describing the Ice architecture summarizes the detailed description found in the Ice manual [20].

Ice clients and servers have the logical internal structure shown in figure 2.3. The Ice core contains a number of libraries which include the run-time support for remote communication. This covers the code necessary for networking, threading, byte ordering, and other networking-related processes. The client and server are linked with the Ice core.

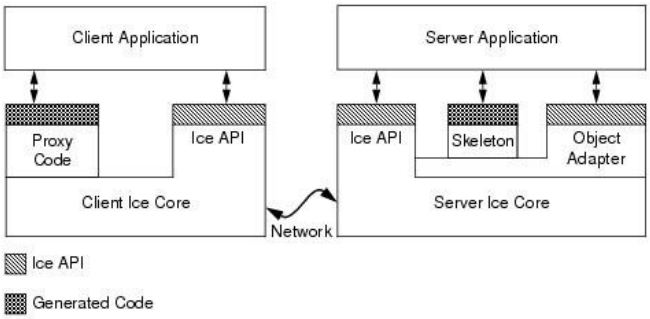

Figure 2.3: The Ice client-server architecture (reprinted from [20])

The Ice application programming interface (API) provides access to the part of the Ice core that is independent of the specific types defined in Slice. The Ice API takes care of chores such as initializing and finalizing the Ice run time. The same API is used for clients and servers.

The Slice definitions are compiled to generate the *proxy* code, which is specific to the types of objects and data that have been defined in Slice. The proxy code allows the client to send a RPC to the server when a function in the generated proxy API is called. The corresponding function on the target object is then invoked. The proxy code also provides marshalling and unmarshalling code.

The Slice definitions will also generate the *skeleton* code, which is the server-side equivalent of the client-side proxy code. That is, it allows the Ice run time to transfer the thread of control to the application code which the developer writes. The skeleton also contains the marshalling and unmarshalling code which allows the server to send and receive parameters sent by the client (and the handling of exceptions).

The server-side also contains *object adapters*. An object adapter maps incoming requests from clients to specific methods on programming-language objects. It is also associated with one or more transport endpoints, meaning that one can associate both a TCP/IP and a UDP endpoint with an adapter. The object adapter is also responsible for the creation of proxies (at the request of server-side objects) that may be passed to clients, since it knows about the type, identity, and transport details of each of its objects. The

object adapters are associated with a *communicator*. The communicator is the main entry point to the Ice run time. It is associated with a number of run time resources such as the client and server-side thread pools, the object adapters, object factories, and more.

2.3.4 Services used

By definition, distributed applications may run on various computers. Each computer is a node and the collection of nodes is a grid. *Ice Grid* adds versatility to distributed applications by providing a naming/location service for clients to query as opposed to having to hardwire the endpoint information (IP addresses and port numbers for example) into the code. Any change in this information would require the application to be recompiled. The port numbers would also need to be manually administered. This quickly becomes a cumbersome task as the number of servers grows. Ice Grid is also used for the deployment of distributed applications by allowing the user to register servers for automatic start-up. In this way, the user does not have to start a specific server before a client makes a request of it. Ice Grid is also used to establish sessions so that only one client can access a given resource (e.g. an object or a server). This is extremely useful when a specific instance of an object is needed for a given client. Ice Grid also offers other services such as load balancing. Currently only the services for naming and automatic start-up of servers are being used.

Other Ice services which facilitate both the development, use, and maintenance of a distributed application include Ice Patch2 and Ice Glacier2. Ice Patch2 is used to allow the maintainers of an application to distribute software updates to clients. Ice Glacier2 is a service which allows clients and servers to communicate securely through firewalls.

2.4 Related work

In this section, we present an overview of robotics software systems (RSS) (as defined in [23]) and relate them to this work.

Software is essential in any robotic system which displays any seemingly intelligent behaviour. The reuse of existing implementations which contribute to this intelligent behaviour (e.g. path planning and navigation algorithms, etc) is not simply desirable, but is in fact required, if any progress towards new research is to be made.

As laid out in [23], RSS tend to fall within one of three categories:

- Driver and algorithm implementations
- Communication middleware
- Robotic software frameworks

Often, the borders between these categories is blurred. Comparisons between RSS within different categories is misleading as each is motivated differently and serves a different function – i.e. they are simply unlike each other except in their shared goal of increasing reusability in robotics. An attempt is made here to present an example RSS from each of the three categories above, and relate them to this work.

The Player project [15] includes a robot device interface which serves as a hardware abstraction layer (HAL) for robotic devices, as well as the 2D robot simulator Stage and the 3D simulator Gazebo. They are all open source and free. Player allows the same interface to be used to control the robot by providing 'drivers' which translate the abstract commands available in the interface into specific commands. The Player Server is an excellent example of the first type of RSS described above (Driver and algorithm implementations).

Middleware for Robotics (Miro) is a distributed object oriented framework for mobile robot control, based on the CORBA technology [19]. The overhead in terms of memory and processing power which results from the use of CORBA is a disadvantage of this solution. In addition, the complexity involved in understanding and learning to use it is also a hurdle. In comparison, the use of the much simpler and more efficient Ice middleware (which is introduced in section 2.3.3) by XPERSIF precludes such problems. Miro is an example of the second type of RSS.

Orca is very much a robotic software framework. It is an open-source framework for developing component-based robotic systems [22]. It uses a component-based approach which allows building-blocks (components) to be developed and used together to create a more complex robotic system. The main motivation is the advancement of robotics research through the reuse of such building blocks. This is done by providing commonly-used interfaces, libraries and a repository of existing components. Initially, Orca used CORBA technology but has since switched to using the Ice middleware at the end of 2005.

While the use of the Player project would address the issue of robot control and perhaps simulation, it could not be used for the integration of a complete robotic system (such as the cognitive architecture presented here). The same can be said of Miro. Of the three RSS presented above, Orca is most similar to XPERSIF in that it is also a thin framework which utilizes the Ice middleware, provides a simple component model and uses simple and efficient communications patterns. An added advantage of the XPERSIF framework is its service-oriented nature. Despite being a mature project, Orca's repository does not provide interfaces, components or libraries which offer the more advanced functionalities relating to cognitive architectures. Chapter 6 presents the

results of the XPERSIF framework and architecture which highlight the advantages of its design.

2.5 Simulation

Robot simulators are widely used in the robotics field for different purposes. They have mainly been used to design and test new robot models as well as to develop the necessary software for running the robots, such as controllers or behaviours. The simulation of multi-robot teams [34], for example, is a vital tool in fields such as Robocup, where the setting up of a whole team of robots is a time-consuming task. Likewise, the simulation can be run for as long as is needed and is not limited by physical constraints such as battery life. In this way, simulation also contributes to speeding up the pace of research. Where multi-robot teams are concerned, a simulator that allows the testing of team behaviours is ideal. A 2D simulator is often sufficient for this purpose. The field of evolutionary robotics also relies heavily on simulation, as the time spans used for their purposes are generally very long. In this special case, a fast simulation is the highest priority.

2.5.1 XPERO's simulation requirements

In the above-mentioned cases, the simulation is used by the researcher to visualize the behavior of the robots and not by the robot itself as is the case in XPERO. Within the XPERO project, the simulation is used by both the researchers and the robot itself. For the robot to function as expected, its perception of its environment should be as realistic as possible, both dynamically and visually. The dynamics of its environment must use accurate models of friction, mass, forces, rigid body collisions, and so on, in order to enable the robot to develop the correct hypotheses. In addition, the dynamics have to allow for an accurate simulation of the manipulation process itself. Realistic visualization of this interaction with its environment is vital for the observation and the perception processes. The robots use a variety of vision techniques and mechanisms such as focus of attention and novelty detection which allow them to find objects to experiment with. In order for these techniques to be used, it is necessary that the visualization be as realistic as possible. The use of lighting, shadows and textures contribute to this realism.

2.5.2 An introduction to the XPERSim simulator

The quality of a simulation is largely dependent on the physics engine which calculates the dynamics of the simulation, and the rendering engine which is used to visualize the simulation. The results of the physics simulation are highly dependent on the accuracy of the models which are provided by the user. There are many physics engines available with varying quality and cost. Similarly, a wide variety of 3D rendering engines, used for

visualization, also exist. The game industry has helped to advance the quality of these engines to its current limits; to the point where open-source engines that provide this exceptionally high-quality visualization are now available.

XPERSim [3] is a 3D simulator, developed by the authors for XPERO, which provides a realistic and accurate physics simulation that is also visually realistic, at a reasonable cost. While many existing robot simulators provide a good dynamics simulation, they often lack the high quality visualization that is now possible with general-purpose hardware. XPERSim achieves this by using the Ogre 3D [29] engine to render the simulation whose dynamics are calculated using ODE [28]. This enables an accurate simulation of robot observation and manipulation tasks. Currently, the Khepera II robot from K-Team has been implemented. A perception module has been developed as proof of concept that allows basic vision algorithms to be performed on the simulated scene. A Client Console (Fig. 2.4) – implemented with a TCP/IP connection – enables tele-operation of the simulated robot using the same Khepera API commands that are sent to the physical robot. In this way, any user with code to control a Khepera can use this code in XPERSim. Simultaneous multiple camera simulation of the rendered scene is possible at high frame rates. A library of components that can be parametrized by the user has been created. This library includes a number of commonly-used sensors and actuators. The flexible parametrization of the sensors will also facilitate the transition from carrying out the experiments within a simulated setting to a physical setting as their error models can be accounted for.

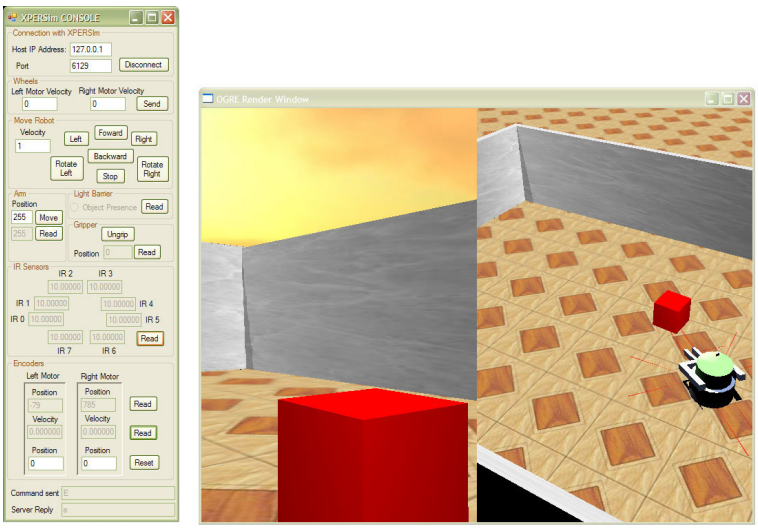

Figure 2.4: A screenshot of an XPERSim simulation alongside the Client Console

XPERSim enables the replication of experiments, regardless of whether the robot is physically present or not. It has been used successfully, not only in the initial stages of the project in allowing the researchers to pursue multiple scenarios simultaneously to develop and evaluate concepts, but also in the later stages by providing vital traces used for the machine learning process. The initial results [6] from the XPERO project support the original assertion that simulation has indeed enhanced the speed of research within the project.

Previous work integrated XPERSim into the XPERSIF framework via Ice allowing it to be run in a distributed setting. While tele-operation and tele-observation of the simulation were implemented, the solution for tele-observation was provided with a focus on fulfilling the use case for data generation which provided traces for the machine learning tools. These traces included the simulated image. While this requirement was met, the solution did not enable a frame-by-frame view of the simulation. The specification of new use cases (presented in the following chapter) specifies the need for the architecture to supply quasi-real time observation of the simulated image. The implementation of the solution is presented in section 5.6.

Chapter 3

THE FRAMEWORK'S REQUIREMENTS SPECIFICATION AND ANALYSIS

As mentioned in Chapter 1, the requirements specification process is the first step in designing a software system. XPERO's middleware and simulation requirements are presented in Chapter 2. This chapter presents the frameworks' requirements as a set of scenarios followed by the specification of those requirements as use cases. It starts with a description of the XPERO scenarios which are key to the formalization of the specification process. A concrete experiment for one of the scenarios, namely for the notion of *movability*, is also included as it contributes to the specification of a use case for the functioning of the whole loop as seen in figure 1.1. The specification use cases of this loop at varying levels of abstraction are then presented.

3.1 The scenarios and showcase

As previously mentioned, this work has been carried out with a focus on achieving the various scenarios which have been developed by the XPERO researchers. These scenarios were developed to explore how the robot might discover certain notions, e.g. containment and movability. The general idea is that the robot's discovery of one notion might lead it towards experimenting to discover another one in a chain of discoveries, often called the *evolution of theories*. The robot here is an abstract robot without a specific embodiment. Following is the series of scenarios that serve as informal specifications [27].

I Firstly, our XPERO robot is first exposed to an empty world. Robot moves around in this environment, but given there is no visual feedback, it will come to the conclusion that its own motion does not have any effect ("I move, nothing happens").

II Secondly, put an object there such as a red ball or a brown box. Robot might discover the concept of ego-motion and its consequences which we subsume under the term naive kinematics ("I move and something happens"). This concept will, however, be tight with egocentric coordinates.

III Thirdly, only if we deploy three or more distinguishable objects in the robot's world, it has a change at all to discover that only it itself is changing the position but the perceived objects do not change their position in the surrounding world. Only with the perception of three or more objects the robot may discover the notion of

allocentric coordinates. Necessary to discover whether or not really all objects in the surrounding environment are fixed, or also change their position, in particular, if they are in contact with the moving robot (notion of movability). Allocentric coordinates are further necessary to discover the notion of an obstacle.

- We place the robot in a closed rectangular room surrounded by four white walls, hoping that it will discover the notion of containment and also what containment means to its existence

IV In a fourth sub-scenario we put a rectangular hole in one of the walls and hope that the robot will discover the concept of an exit, and what the exit means for its existence, its desires and intentions

V Finally, the robot should be finally able to cope with one or two boxes placed in front of the exit of the otherwise empty room. Or else we have to go back and refine or redefine the concepts and notions or also consider other learning paradigms which may not be entirely unlikely.

The XPERO showcase is defined in the final scenario above. While these scenarios explain the setting for each notion, there is no detailed explanation of actions which the robot might take to discover the notions. In order to specify the software requirements as use cases a series of mind experiments were performed (the results of which may be found in section 6.4.1). The outcome of these experiments is a set of partial plans that might be combined and executed to help achieve the behaviours mentioned in the scenarios above. Use cases are useful in capturing the functional requirements of a system. They focus on what the system must do as opposed to how it will accomplish the tasks.

Additionally, a detailed experiment specification exists for the notion of movability – the movability of an object triggered by the robot's own actions. This experiment, although currently performed with the aid of a human researcher in the loop providing the plan, is used as a specification of the fully functioning loop.

Within this concrete experiment for the movability scenario, the robot is located within an environment containing four boxes of varying color. Two of these boxes can be moved by the robot whereas the remaining two boxes are too heavy for it. It starts out with a notion that all boxes are not movable. The robot performs default tasks such as roaming around its environment comparing observations to predictions. Once the robot is surprised it attempts to validate this surprise to ensure that it was not noise that triggered it. This is done by attempting to recreate the conditions within the environment before it was surprised and repeating those actions that triggered this motivation. Once this has been achieved, it then designs a series of experiments to

collect positive examples of the robot attempting to push boxes. Once it has collected a pre-specified number of positive examples, they are used in the learning process. The use case for this single experiment spanning the loop is found in section 3.2.3.3.

3.2 Formalization of the specifications

In this section we formalize the specifications presented above by identifying the actors involved, generating a state diagram and specifying use cases.

3.2.1 The learning loop

As mentioned in Chapter 1, the XPERO learning loop can be seen as an implementation of the scientific method to be performed by an autonomous embodied agent. For each process involved in this method an actor(s) is identified. These actors can be seen alongside the processes they perform in figure 3.1 and an overview of each component is presented below.

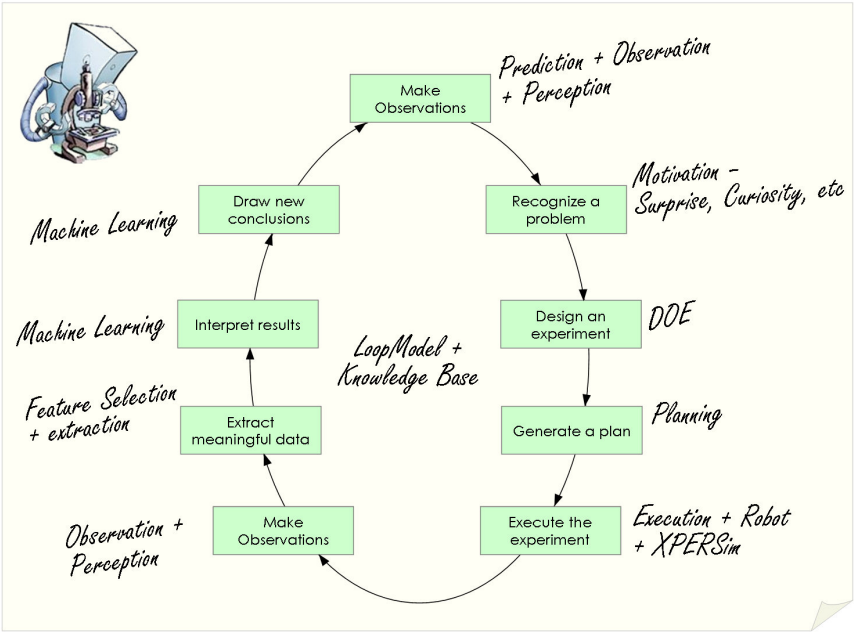

Figure 3.1: The experimental loop annotated with the relevant XPERSIF components.

The effective design of experiments (DOE) is a topic which has been addressed by many researchers as in [12]. It allows useful observations to be made. Its output is a coarse-grained experimental concept.

Planning and execution are well known fields in robotics. Given an initial state of the world and a goal state, a planner generates a plan that should allow the goal state to be

reached from the initial state [16]. Within the learning loop, the initial state is the one that the robot is currently in, and the goal state is obtained from the DOE component.

Due to the dynamic and uncertain characteristics of the environment as well as the need to coordinate the various hardware components of a robot, plan execution as a component within the loop is a necessity. The nondeterministic nature of the results of any action taken by the robot necessitates that the plan's execution be monitored. This allows errors that might arise to be detected and handled while also allowing the detection of opportunities to execute partial plans.

In observing its environment, a problem arises in that the robot is bombarded with a huge amount of information from sensors. Despite this, it usually has an uncertain and incomplete view of its world due to noise and to its embodiment and position within the world. Just as humans extract features from the environment, such as edges, faces, and objects, a robot must do the same in order to extract meaningful data from its observations. This functionality is provided by the feature extraction component.

Selecting which features must be observed and measured is as important as designing a good experiment. This functionality is encapsulated into the feature selection component.

The feature processing component uses the extracted features such as objects and their poses to compute 'redundant' features such as distance between two objects, for example. It can thus be seen as high-level feature extraction. It is also responsible for creating feature vectors from the selected features which are sent to other components which need to monitor them.

Motivation stimuli such as hunger, surprise and curiosity play extremely important roles in human (and animal) behaviour. In much the same way, artificial forms of these stimuli are used to propel the robot to adopt various behaviours. A robot's curiosity about a region of its environment might cause it to explore it. Surprise at an observation that does not match a prediction might cause it to experiment in order to explain the observation. Similarly, hunger might drive the robot to find an exit to the room within which it is contained in search of a power supply.

What we know about the world allows us to act within it and react to the changes around us. Similarly, what the robot knows about its environment, either from *a priori* knowledge placed there by researchers or by discoveries it has made about the way the world behaves, enables the robot to act and react to its world. Naturally, this model of the world is often inaccurate and it is at those moments when the prediction of behaviour is incompatible with the observations made that surprise is triggered.

The machine learning module aims to induce from the observed data interpretable theories that could enable not only predictions in the robot's world, but also reasoning about the domain, and possibly the gaining of new insights [5].

Memory, both long term and short term is a necessity in any learning process, and is provided by the knowledge base.

3.2.2 Loop states

In analyzing the informal description of the movability experiment, the system can be seen as being in one of the four states shown in the state diagram in figure 3.2.

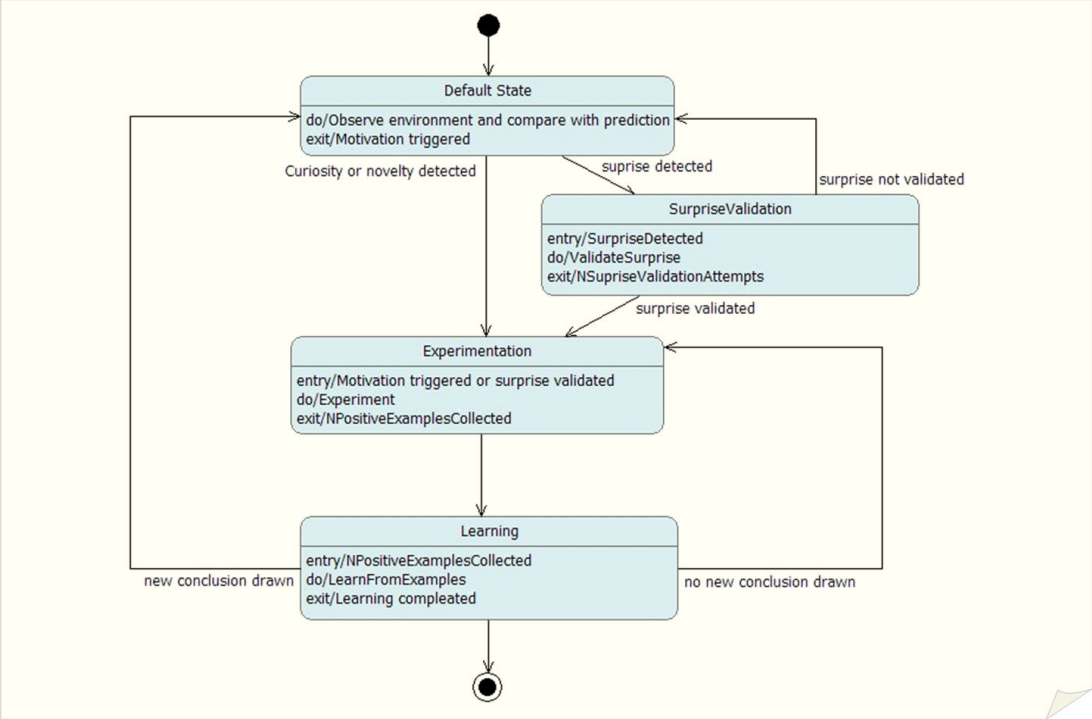

Figure 3.2: A state machine diagram depicting the states that the system can have while performing the movability experiment, and the transitions between them.

It should be noted that the state diagram shown in figure 3.2 is the direct result of the analysis of the existing experiment for the movability scenario. Naturally, other experiments may result in different state flow diagrams (which may include new states). The flexibility and extensibility of the framework and architecture in handling such developments is addressed in Chapter 6.

3.2.3 The use cases

The following use cases will be used to evaluate the framework as a whole and in parts as they relate to a robot learning experiment in addition to individual subsystem behaviour. As such, they represent the system at two different levels. The use case of the experimental loop treats the system as a whole and does not identify any additional actors other than the human researcher. On a more specific level, actors that perform various tasks within the loop are identified and their specification is then presented in the consequent use cases.

3.2.3.1 Use case for feature selection, extraction and processing

The perception of the environment (following its observation) and taking measurements is carried out by the feature selection and extraction actors. It is achieved by obtaining the sensor readings available to the embodiment, selecting features which need to be extracted, extracting basic features such as objects and their poses before processing them to calculate higher-level features such as distance or velocity and finally generating the feature vectors. These processes can be seen in figure 3.3.

Use case name:	Make observations
Goal:	Make observations of the environment in order to generate feature vectors.
Summary:	The robot makes observations of its environment selecting the appropriate high level features and extracting them from sensor data.
Actors:	Robot
Preconditions:	The loop is running and the system is receiving sensor data.
Triggers:	The hardware components are ready.
Basic course of events:	1. The system selects appropriate features to extract. 2. The system receives the sensor data from the embodiment (and the overhead camera if available). 3. The system extracts basic features from the sensor data. 4. The selected high level features are calculated using the different low level features. 5. The feature vector is generated and distributed to subscribers.
Post conditions:	The subscribers receive the feature vector.

3.2.3.2 Use case for plan execution

When a plan is executed, three basic processes always take place: sending commands to the embodiment for execution, monitoring this execution to ensure that it proceeds as expected (to effectively handle errors as they arise, and to detect opportunities) and finally making observations of the environment through the various sensors available to

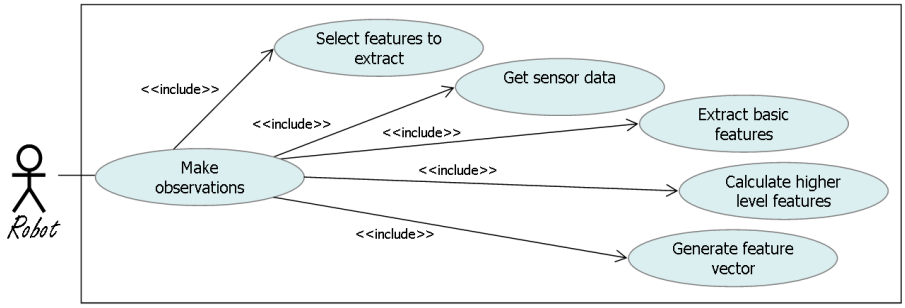

Figure 3.3: The use case for feature selection, extraction and processing.

the robot (including external sensors such as overhead cameras). This can be seen in figure 3.4. A series of partial plans are the outcome of mind experiments (see section 6.4) performed on the scenarios mentioned in section 3.1. While the scenarios consider an abstract robot, these partial plans assume a mobile manipulator (while remaining flexible to handle specific embodiments fulfilling this criterion). The use case diagrams for these partial plans may be found in Appendix D. The listing is non-exhaustive and can be extended easily. A detailed specification of algorithms which would implement the use cases is found in section 6.4.1.

Use case name:	Execute plan
Goal:	Execute a given plan
Summary:	The robot executes a previously generated plan.
Actors:	Robot
Preconditions:	1. A plan is generated. 2. The robot (and overhead camera) hardware is ready 3. The initial state of the plan is the current world state.
Triggers:	The sending of the plan for execution.
Basic course of events:	1. The robot decomposes the plan into commands and operations (as in the recipes presented in section 6.4.1). 2. The system sends commands to the robot and queries it (and the overhead camera) for information. 3. The system monitors the execution of the commands. 4. During execution the robot makes observations of the environment.
Alternative paths:	If the execution process fails, the robot might replan or return to the default state or in the case of a failure in the hardware (or software), the whole experiment might end.
Post conditions:	The plan is executed.

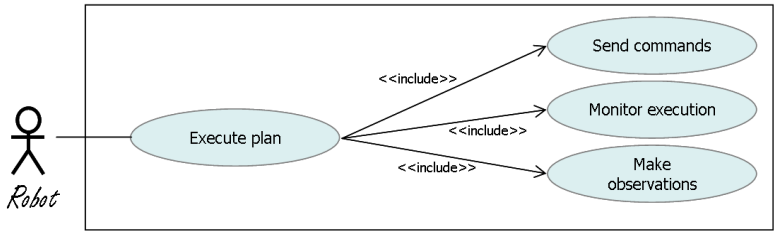

Figure 3.4: The use case for plan execution.

3.2.3.3 Use case for the loop

In actuality, there exist two users for this software. One is the human researcher as is currently the case and the other is the robot itself. While the specification of the requirements has been done for the human in the loop, the case for the robot as the user is easily accomplished with the integration of the fully functioning software provided by the partners. This use case specifies the scenario for the movability experiment as outlined in section 3.1. It describes the interaction between the human client and the system represented as a sequence of simple steps. This use case is comprised of many smaller use cases. There are four main use cases which correspond to the four states of the concrete movability experiment. As mentioned previously, this is one sample flow of the loop and alternative courses are of course possible. These four main use cases are also presented in this section.

3.2.3.4 Use case for distributed simulation

A use case specifying the process for distributed simulation is shown in figure 3.6. Here a human researcher is the actor who uses the TeleSimView client detailed in section 5.6 to observe the experiment from the viewpoint of the robot and/or overhead camera.

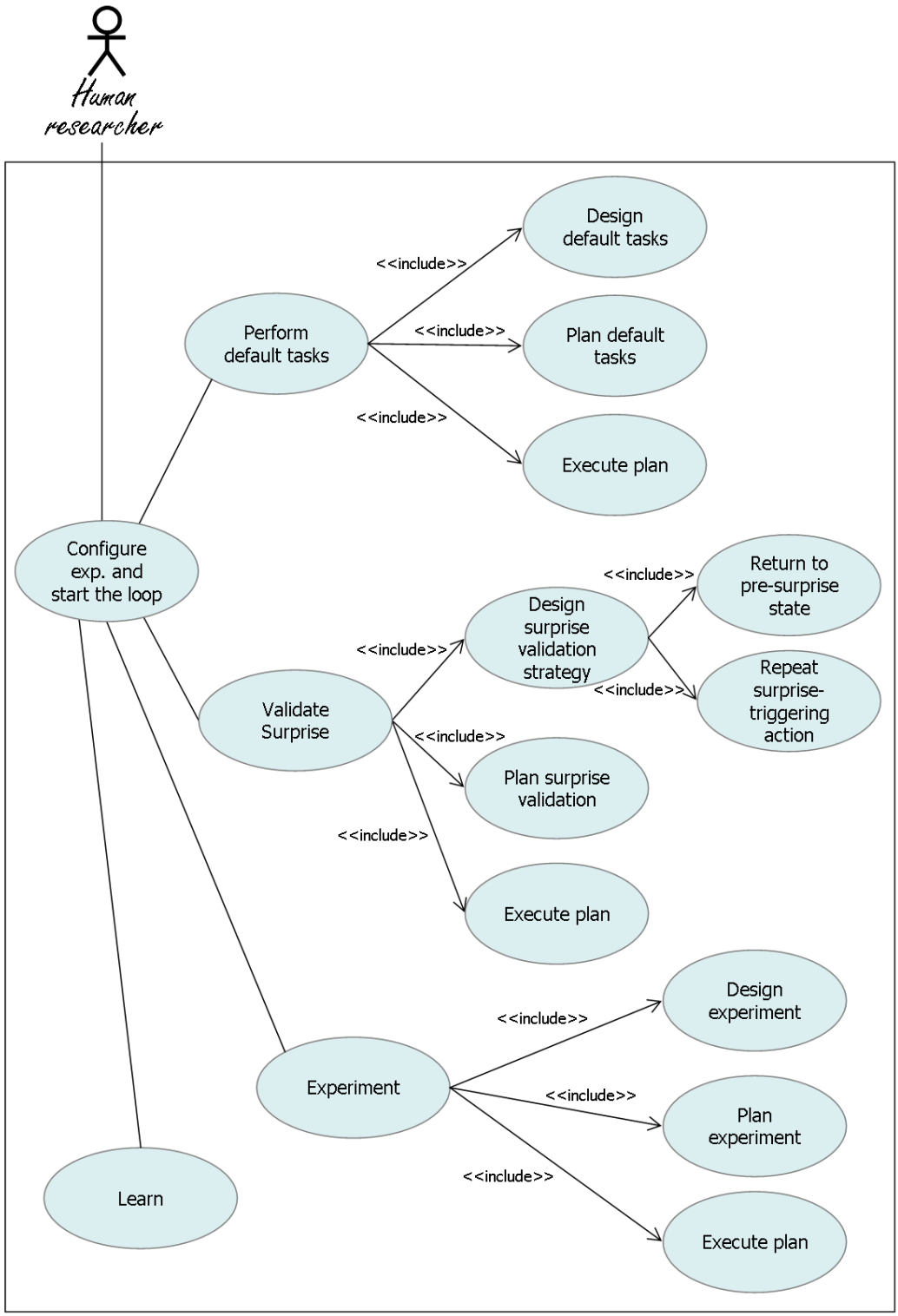

Figure 3.5: The use case for the movability experiment.

Use case name:	Loop using surprise as stimulation of experiments (Movability experiment)
Goal:	Configure the cognitive loop and start an experiment where the ultimate goal is that the robot learns by experimentation when surprise is detected.
Summary:	In a configured environment, a robot performs default task until surprise is detected, causing it to first validate the detected surprise and then design and perform experiments to explain it and learn new conclusions.
Actors:	Human researcher
Preconditions:	The loop is configured.
Triggers:	The loop starts.
Basic course of events:	1. The human researcher configures the loop and starts the experiment. 2. The robot (inside the system) chooses and performs default tasks. 3. During the execution of these tasks, it makes observations of the environment. 4. It compares the observations with the prediction models in the knowledge base. If there is a discrepancy between the predictions and the observations, surprise is detected. 5. If there is a discrepancy between the predictions and the observations, surprise is generated. 6. It then designs and executes a strategy to validate the detected surprise. 7. If it manages to validate the surprise motivation, it designs and performs an experiment that explains it. While executing the experiment, measurements are made in the form of observations. 8. It interprets the observations made while experimenting to draw new conclusions and add new knowledge to the knowledge base.
Alternative paths:	If surprise is not validated, the robot returns to step two.
Post conditions:	The learning process is completed (successfully or unsuccessfully).

Use case name:	Perform default tasks (State 0)
Goal:	Observe the environment in search of motivation.
Summary:	As the environment is static, the robot must perform actions within it in order to vary its observations. The robot selects a strategy for collecting observations. This might simply be to roam. Alternatively, it may perform random actions that are not limited to its mobility. It then performs these actions, all the while making observations and attempting to recognize a problem and spur the design of an experiment.
Actors:	Robot
Preconditions:	The loop is configured and ready.
Triggers:	The loop is stared.
Basic course of events:	1. The robot chooses from a set of default action strategies. 2. It generates a plan to perform these actions. 3. It performs these actions. 4. It makes observations while performing the actions. 5. It processes the observations in an effort to motivate an experiment (if the observations trigger surprise).
Alternative paths:	One cycle through this system is complete and surprise was not triggered. In such a case, the robot would start with step one again and continue through the basic course of events.
Post conditions:	An experiment is motivated.

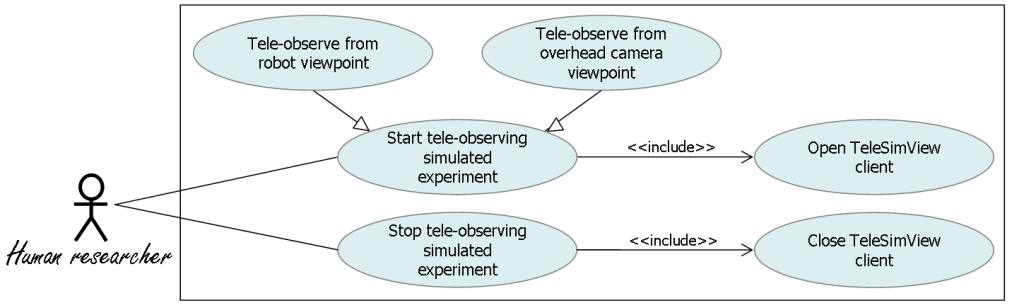

Figure 3.6: The use case for tele-observing a simulated experiment.

Use case name:	Validate surprise (State 1)
Goal:	Validate that the surprise which was previously detected is genuine (and not the result of noise, etc).
Summary:	The robot first attempts to recreate the scene (the environmental settings and itself) just prior to when the action which triggered the surprise was performed. Once this has been achieved it attempts to repeat the action which previously produced the surprising observation. It does this a number of times and only after a consistent triggering of the surprise stimulus does it consider the validation complete.
Actors:	Robot
Preconditions:	A surprising observation was made.
Triggers:	Surprise was detected.
Basic course of events:	1. The robot designs a surprise validation strategy (this is a predefined strategy within the movability experiment where the robot recreates the scene prior to the detection of surprise and repeats the action which produced the surprising observation). 2. It generates a plan to carry out this strategy. 3. It executes the plan. 4. It makes observations while performing the actions. 5. It uses them to check for surprise again.
Alternative paths:	One cycle through this system is complete and surprise was not validated. In such a case, the robot would return to state 0 (Perform default tasks use case presented above).
Post conditions:	A surprise validation strategy is executed.

Use case name:	Experiment (State 2)
Goal:	Design and perform an experiment that explains the surprising observation.
Summary:	The robot designs an experiment, plans it and then executes it and making observations.
Actors:	Robot
Preconditions:	The robot was motivated to experiment.
Triggers:	A motivational stimulus (surprise in this concrete experiment).
Basic course of events:	1. The robot designs a well-defined experiment using the surprising observations. 2. It generates a plan to carry out this experiment. 3. It executes the plan a certain number of times in order to collect a pre-defined number of positive examples (20 in the concrete movability experiment). 4. It makes observations while performing the actions.
Alternative paths:	The execution may have failed in which case the robot might attempt to execute the same plan again or return to the default state or design another experiment.
Post conditions:	The designed experiment was performed a predefined number of times and a collection of observations have been made.

Use case name:	Learn (State 3)
Goal:	Interpret the results of the experiment
Summary:	The robot uses the observations made during the experiment to learn.
Actors:	Robot
Preconditions:	The robot made observations during an experiment.
Triggers:	The completion of the observation process during an experiment.
Basic course of events:	1. The robot chooses a learning method. 2. Format the observations for the chosen method. 3. Interpret the observations using the chosen method. 4. Analyze the results to see if they successfully explain the surprise. 5. Draw conclusions and ad them to the knowledge base.
Alternative paths:	If the learning process fails to explain the surprising observations, a new set of features may b selected and the experiment repeated or the robot may return to performing default tasks.
Post conditions:	The learning process is completed (successfully or unsuccessfully).

Use case name:	Tele-observation
Goal:	Tele-observe a simulated experiment running on the server.
Summary:	A researcher uses a tele-observation client to observe a simulated experiment from the viewpoint of the robot and the overhead camera.
Actors:	Human researcher
Preconditions:	The experiment is running on the server.
Triggers:	The human researcher wishes to observe a simulated experiment.
Basic course of events:	1. User opens the tele-observation client. 2. The tele-observation client subscribes to receive simulated images. 3. The system transmits simulated images to the tele-observation client. 4. The human researcher tele-observes the experiment currently underway in a quasi-real time manner. 5. The human researcher stops the observation at any time by closing the tele-observation client. 6. The tele-observation client unsubscribes from receiving images and closes.
Post conditions:	Tele-observation window is closed and the experiment's execution is not affected by the tele-observation process.
Issues:	What about the physical setting?

Chapter 4

THE SOFTWARE INTEGRATION FRAMEWORK AND ARCHITECTURE: XPERSIF

In this chapter, the design of the framework and architecture is presented through the specification of the component model which forms the basis for all components within the loop. The way in which components communicate with one another plays a large role in the system's design. As such, the use of communication patterns within the architecture is summarized. The data flow between components is then presented followed by the control flow in the various states of the loop.

Components within the XPERSIF architecture may be classified into one of three basic groups of components (depicted in figure 4.1).

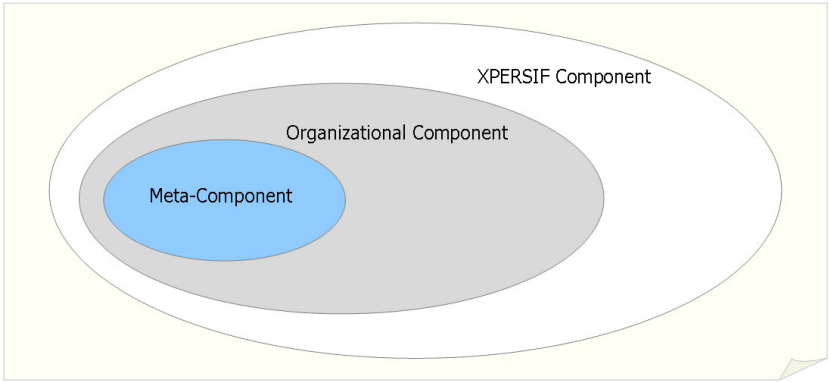

Figure 4.1: The classification of XPERSIF components.

The XPERSIF component model provides the basic component model for all components within these three groups:

- *basic* components
- *organizational components* responsible for managing a hierarchy of components.
- *meta-components* organizational components which appear as a single component while internally, they are composed of individual components which cooperate to provide the services of the meta-component (as presented in figure 4.2).

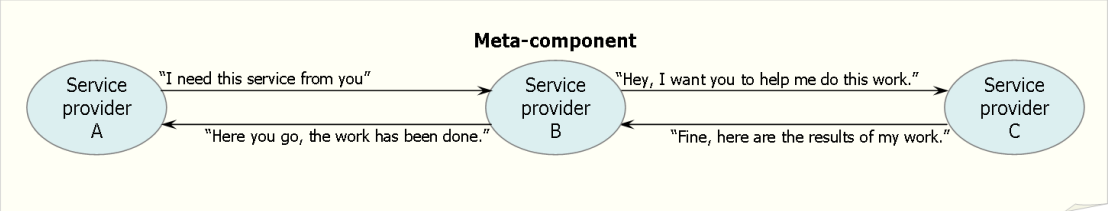

Figure 4.2: An illustration of components cooperating to provide the services of the meta-component (based on the diagram in [11]).

4.1 The XPERSIF component model

A component model provides a standard for developing components within a framework. This facilitates the development process. In this section, we present the architecture through an example component.

A component's structure can be summarized by stating that it offers services as commands and operations (as defined in section 4.1.2) and notifies (as in section 4.1.3) its users of the final state of the service. This can be seen in figure 4.3. In order to provide this functionality several abstract interfaces are specified. Namely, the Operation interface (which provides functionality for component administration), the Subject interface (which provides a means to subscribe to notifications) and the Observer interface (which provides a means to receive notifications). The component-specific functionality is specified within the Component interface. As none of the components within the system are hard real time components, this component model meets the soft real-time requirements in the simplest way possible. Operations are used for services which complete quickly, commands for those that need more time, and notifications serve to enable a call to a command to quickly return (thus enabling a non-blocking call) and to provide a monitoring and error handling mechanism. These three interfaces are sufficient for the requirements of the project.

The first step in integrating a new component is to specify it in the Slice language. A Slice module for the EXAMPLE component is created: **module** Example in a file named `Example.ice`.

```
module Example{
};
```

Next, the definition of the set of operations and commands that implement the functionality of the component must be specified. These are contained within an interface

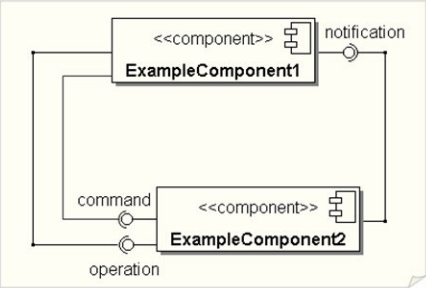

Figure 4.3: The XPERSIF component model.

of the component. The interfaces show what commands exist for a given component and how they may be called (what arguments they take, etc). Interfaces can be thought of as a contract. Once the interface is specified, its implementation can proceed. The actual implementation is transparent to the client - it is just the interface that is visible and the information within it is sufficient for the components functionality to be used.

In the architecture, a module has a number of interfaces. The specific functionality of a component is placed within the IExample interface of our example component. Here, commands and operations are listed with their return types, argument types and variable names.

```
module Example{
    interface IExample extends XPERSIF::IOperation{
    };
};
```

The interface extends the IOperation interface which contains operations that all components use, e.g. starting the component, exiting it, and returning a list of running tasks. This means that in addition to implementing the functions for the commands and operations specific to the example component, the functions in the IOperation interface must also be implemented. The IOperation interface is located in the XPERSIF module (section B.1). The contents of this interface are described next.

4.1.1 Basic properties

As mentioned previously, all components, whether they encapsulate the functionality of software or hardware systems, must extend the IOperation interface within the XPERSIF module and implement a number of common operations. These operations allow the component to be started, braked, continued and exited. They also allow the component's state to be queried by any other component. An operation for obtaining the state of a running command as well as another operation which returns a list of

running commands (should the component be capable of executing two commands in parallel) are also part of this interface.

There exists an extended model for the organizational components. These commands and operations may be found in `XPERSIF.ice` (section B.1) within the IOrganizationalOperation interface. These provide functionality which allows them to manage a hierarchy of components which they are responsible for. These components are detailed in section 5.2. An operation allowing these components to set the state of any given component is also implemented in the IOperation interface.

4.1.2 Commands and operations

It is imperative that a component never blocks. To achieve this, while at the same time allowing for tasks which may have varying duration, a differentiation is made between commands and operations. Both commands and operations return immediately, however, commands will start an asynchronous process which completes when the component notifies the original caller of the command's completion. Commands are used for tasks which may take time to complete, such as moving to a position for example or gripping an object. They are implemented as non-blocking RPCs. From a planning perspective, commands could be seen as planning operators and as such should have preconditions and effects so that a planner can make use of this information. In the Slice definition of a component, it may be useful to specify these preconditions and effects within the comments for each command. These preconditions and effects may be seen as a contract. Operations on the other hand do not start an asynchronous process. Tasks such as getting the readings from the IR sensors or setting the maximum velocity are implemented as operations as they take very little time to execute and pose no risk of blocking the component when they are called. This distinction has many effects on the architecture. It may be useful to specify preconditions and effects for operations as well. For example, an operation which should deliver the shape of an object might specify as a precondition that the object is in view at a specified distance from the camera. The definition of such preconditions for operations would then form a contract (as with commands). In the code snippet below the return types of the functions within the IExample interface reflect this distinction.

```
module Example{
    interface IExample extends XPERSIF::IOperation{
        XPERSIF::ReturnCode exampleOperation(out double variable1);
       /**
         * What the command does
         * @Command
         * @precondition A precondition of the command
         * @precondition Another precondition of the command
         * @effect An effect of the command
```

```
        *
        * @param variable1 in: Description of variable1
        * @param variable2 out: Description of varable2
        * @return CommandReturnCode
        */
        XPERSIF::CommandReturnCode exampleCommand(int variable1, out int variable2);
    };
};
```

The ReturnCode (seen in figure 4.4) is a data structure which is generated by the implementation of an operation within XPERSIF and is returned immediately to the client which called the operation. It is defined within the `XPERSIF.ice` file, along with other XPERSIF data types. It is composed of a ReturnState - an enumeration of possible states which the operation could have returned with; and a Description which may be left empty or may contain an error message. If preconditions and effects have been specified for the operations, a violation of this specification would result in a ReturnState of 'ContractViolation'. A violation of the preconditions would cause the component implementing the operation to immediately return that state. If the preconditions are met and yet the effects are not met, this state is also returned. The state 'Abort' is usually a result of external effects upon the component and signifies that the task could not be performed but that the component is in a safe state. This is not the case when the ReturnState is 'FatalError'. This state signifies that the task could not be performed and that the state of th component is unknown.

```
struct ReturnCode {
        ReturnState state;
        string description;
};
```

```
enum ReturnState {
        Success,
        Abort,
        ContractViolation,
        FatalError
};
```

Figure 4.4: The ReturnCode and ReturnState data types

The CommandReturnCode which can be seen in figure 4.5 is returned immediately to the client by the implementation of a command. It contains a ReturnState and Description as did the ReturnCode above, in addition to another structure, specific to commands, called the CommandTaskID which provides a system-wide unique identification of a specific call to a command. This CommandTaskID includes the ComponentID (a unique identifier for each component, defined in `XPERSIF.ice`), the CommandType (a unique identifier within each component for each possible command), and the Number (the number of times this specific command has been called). Along with DurationToFinish which specifies the time foreseen to execute a command. These items form the CommandReturnCode.

```
struct CommandReturnCode {
        ReturnState state;
        XPERSIF::CommandTaskID commandTaskID;
        string description;
        long durationToFinish;
};
```

```
struct CommandTaskID {
        short componentID;
        long number;
        long commandType;
};
```

Figure 4.5: The CommandReturnCode and CommandTaskID data types

The commandTaskID thus enables the users of a component to query the status of each command and the state of all components at any given time. It also enables the users to be notified when a command first starts and when it completes (either successfully or unsuccessfully) as it is part of another data structure used by the notification mechanism.

4.1.3 Notification mechanism

Any given component may receive notifications of command execution statuses from any other component by using the observer design pattern to observe that component. The functions necessary for this process are placed within the ISubject interface which in this case is called IExampleSubject. This interface includes the two functions below for subscribing and unsubscribing components as observers.

```
module Example{
    ...
    interface IExampleSubject{
     XPERSIF::ReturnCode attachObserver(IObserverOfExample *observerExampleRef);
     XPERSIF::ReturnCode detachObserver(IObserverOfExample *observerExampleRef);
    };
};
```

In order for the component to notify its observers of any change in the status of a command's execution, it calls an operation which is contained in its IObserverOf interface (as seen in the Slice snippet below). Any component wishing to attach itself as an observer needs to extend the specified IObserverOf interface found within the observed component's Slice definition. Through this typing of each component as an observer of the other, the dependencies within the static architecture are made explicit. The architecture is seen within the typing system itself.

```
module Example{
    ...
    interface IObserverOfExample{
        XPERSIF::ReturnCode updateCalledByExample(XPERSIF::CommandTaskInfo comamndTaskInfo);
    };
};
```

The CommandTaskInfo structure shown in figure 4.6 is passed as argument to all observers. It contains, in addition to the CommandTaskID, a CommandTaskState enumeration of possible states which the command may have, such as 'Running', 'Finished' or 'CommandAbort'. It also contains a Description and a DurationToFinish (as defined above). Thus, the observers have complete knowledge of the state of execution of a command. They may also query the component for the state of execution of the command using the CommandTaskID structure as argument.

```
struct CommandTaskInfo {
        XPERSIF::CommandTaskID commandTaskID;
        CommandTaskState state;
        string description;\
        long durationToFinish;
};
```

```
enum CommandTaskState {
        Running,
        Finished,
        CommandAbort,
        CommandFatalError
};
```

Figure 4.6: The CommandTaskInfo and CommandTaskState data types

Finally, the interface IExampleComponent extends the above interfaces, with the exception of the IObserverOfExample interface. This allows for an easy implementation of the component by providing only one class to extend.

```
module Example{
    ...
    interface IExampleComponent extends IExample, IExampleSubject{
    };
};
```

With this, the Slice definition of the component is complete. The file `Example.ice` is compiled to generate all the information necessary to use the functions. So far, the Slice definition has been platform and language-independent. At this point, the user compiles the Slice definition to create a platform-specific and language-specific implementation of the module. In the implementation presented here, the mapping between Slice and C++ is used thus creating a source and header file for the windows environment. The `Example.ice` file would generate `Example.cpp` and `Example.h`. These would be placed with the source and header files of both the client project and the server project.

On the client-side, a connection is established with the component. Using the specification in the interface, the commands and operations are called. The client should also be notified of the commands execution status. A class **ObserverOfI** is implemented as part of the client's project. This class extends the IObserverOfExample interface allowing it to receive notifications.

On the server-side, a class **ExampleI**, which contains the implementation of IExampleComponent should be created. The header file of this class must include the

generated header file `Example.h`. This implementation class must implement all the functions of the interfaces which IExampleComponent extends.

The distinction between commands and operations is also present within the implementation class **ExampleI**. As commands take time to execute, threads are used for their execution to prevent blocking. When an example command is received, a commandTaskId is assigned to it. A thread is spawned to execute the command and notify observers once it completes. For the thread's implementation, a class, called **ExampleCommandStatus** for example, is created using the abstract base class **Thread** from Ice. After the thread is spawned, the *notifyObservers* function within the component notifies the observers which are attached that a new command is running. Finally, a CommandReturnCode is returned to the client to confirm the acceptance (or the rejection) of the command. Once the command has finished (successfully or not), the thread calls *notifyObservers* to notify observers of the command's status. The sequence diagram in figure 4.7 shows the notification process in more detail.

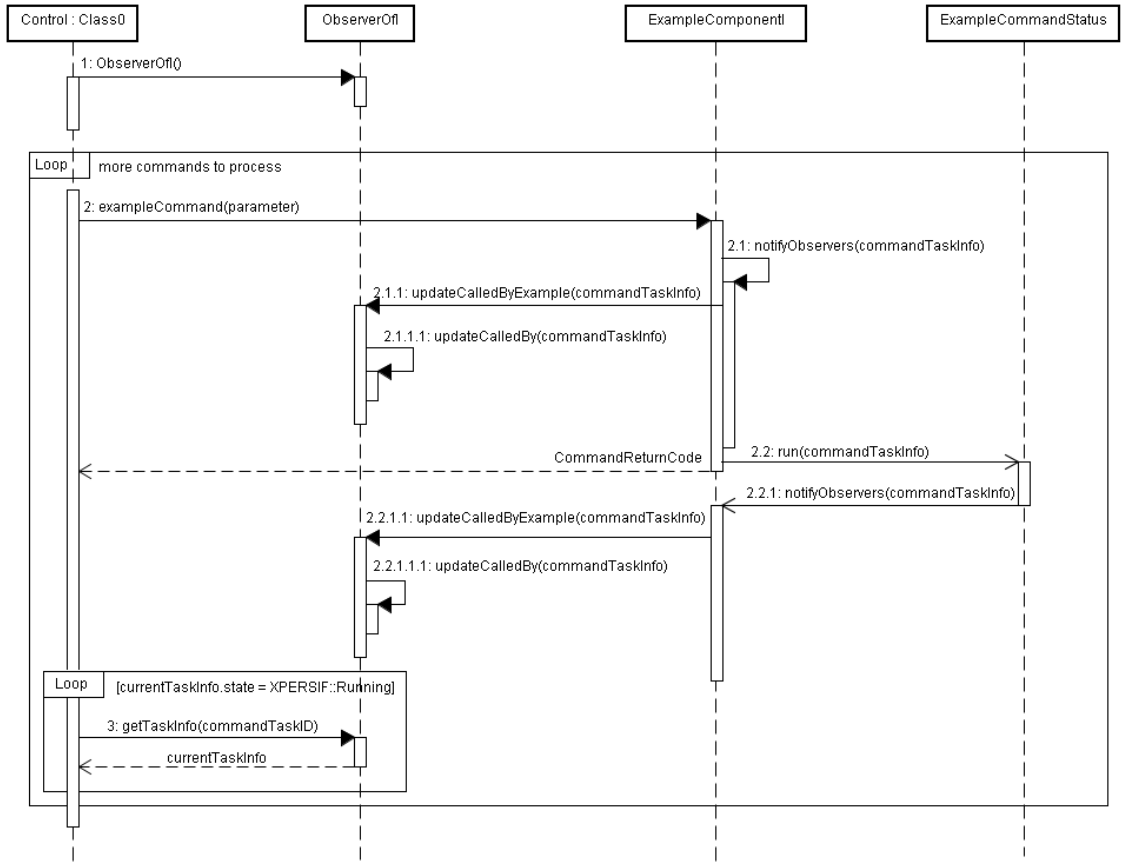

Figure 4.7: A UML sequence diagram showing the notification process in XPERSIF

The contents of the Example Component package are depicted in figure 4.8.

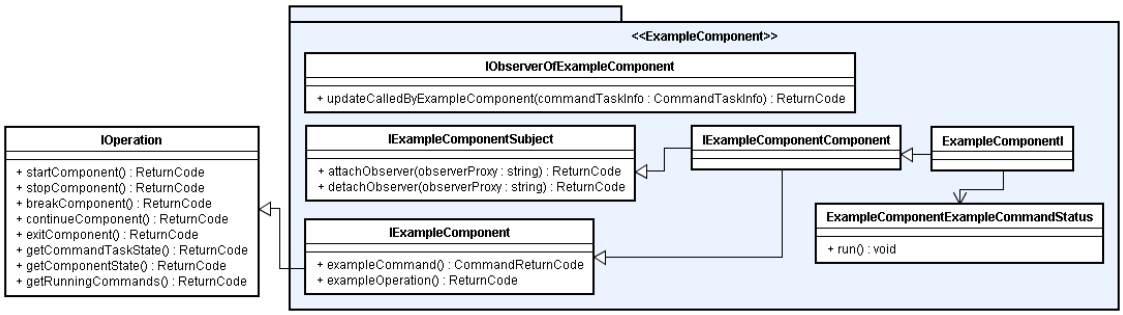

Figure 4.8: The contents of the ExampleComponent package

4.2 Configuration / initialization

As mentioned previously, there exist organizational components whose function is to orchestrate and monitor components. In the implementation found in this work, there exist two such components – LoopModel and RobotModel. These components are detailed in section 5.2. For the sake of detailing the configuration and initialization sequence of components, we represent the special organizational component with the XPERSIFModel component. For this explanation, we focus on a single set of components, the XPERSIFModel organizational component and a single client.

XPERSIFModel starts, observes and shuts down a set of components. It provides a focal point for any component to query the status of this set of components. Within this hierarchy, it is the first to be started and the last to be stopped. The remaining components must register their status with the XPERSIFModel upon registering with the naming service on start-up, and will thereafter continuously update their state with XPERSIFModel.

Figure 4.9 shows a UML sequence diagram for the start-up process of a single application server and a single component it is responsible for. The client starts the XPERSIFModel component which sets the components' state to 'Unknown'. It starts the basic component and once this basic component has registered with the XPERSIFModel component, this state is changed to 'Registered'. After the steps needed to initialize the component are executed successfully, the component relays this change to XPERSIFModel, setting its component state to 'Initialized'. Finally, the component may attach itself as an observer of any other component (it may be necessary for the component to wait until the component it wishes to observe is registered). Having successfully done this, the component once again notifies XPERSIFModel by changing

its status to ‘Ready’. Once components within its hierarchy are in the ‘Ready’ state, XPERSIFModel itself sets its own state to ‘Ready’ and commands may be accepted.

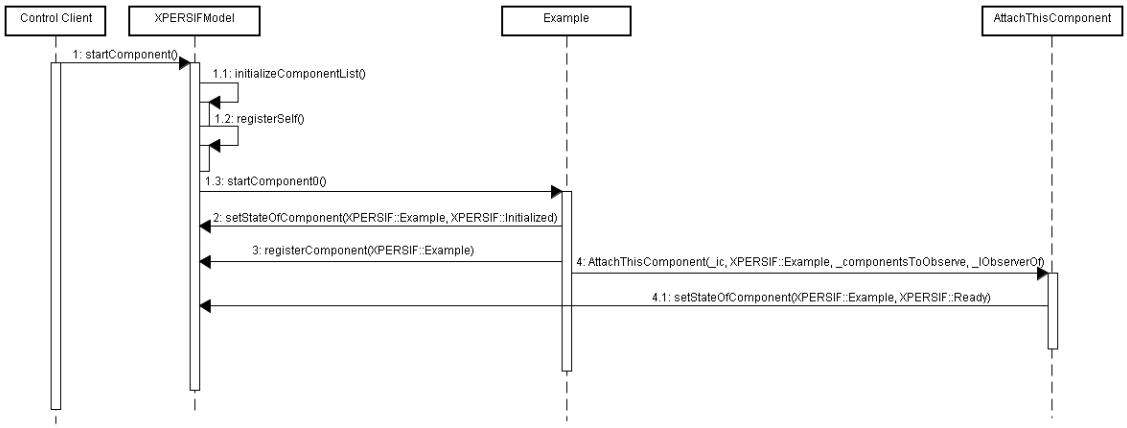

Figure 4.9: A UML sequence diagram showing the start-up and initialization process for the components.

The component contains only one command: *monitorComponentState* which takes a ComponentState as argument and is used to notify the client, for example, when all components which register with the given organizational component have the same ComponentState. The client, for example, calls this command with the ComponentState ‘Ready’ after starting all components before starting to send any commands/operations.

The operations *setStateOfComponent* allows the components to set their state. The *getStateOfComponent* delivers the state of a given component. Register component is called by the individual components to register with XPERSIFModel. The *getAllComponentStates* delivers a list of all the components and their component states. These functions are implemented by all organizational components and are found within the `XPERSIF.ice` file.

4.3 Application of communication patterns in component integration

As mentioned in section 2.2, the choice of communication patterns within an architecture plays a crucial role. There is often more than one good solution to a given problem, and careful consideration must be given in order to provide the optimal one.

The observer design pattern is used by components to observe one another. This has been implemented using the publisher-subscribe communication pattern described in section 2.2.

The same method is used to subscribe to receive well-defined, parametrized features of the observations made by the robot. Similarly, the pattern is used to subscribe to the

tele-observation service which allows the human researcher to view an experiment from the robot and overhead camera view. By building this notification system into the framework, it becomes middleware-independent and allows the framework to remain thin (as in [23]) and easily reused.

Within certain components, such as the robot perception component, a blackboard architecture [2] is set in place, allowing raw sensor readings, for example, to be obtained from the robot at a certain frequency and made available at a central location for all the component's functions to look up on demand.

A loose coupling between the services of a SOA is a key principle. In an effort to maintain this loose coupling in the XPERSIF framework, components always communicate with each other through their interfaces even when the given components reside on the same server. This ensures that the component architecture, once implemented, can be easily altered.

4.4 Data flow within XPERSIF

A component diagram depicting the data flow between the various components of the loop is shown in figure 4.10. It is the result of the requirements analysis process. It shows each component's input and output as data flow. The components here have been grouped according to their functionality. This diagram does not elaborate on the implementation of each of these components – for example, it does not mention which are applications and which are components that are grouped to form an application. They simply show the components of the loop and the data flowing between them. The implementation specifics are detailed in Chapter 5.

The LoopModel component which is the organizational center of the loop is seen at the center of figure 4.10. It serves as an entry point to the loop for a graphical user interface (GUI) (or a console client as in the current implementation) which uses it to configure the loop. It is responsible for parametrizing, starting, monitoring and exiting the necessary components within the loop. For the sake of simplicity, the flow of the component status information from each component to the LoopModel is not depicted. This component is detailed in section 5.2.1.

The GoalDesign module is responsible for the robot's actions while it is in the default and surprise validation states. For example, it might choose to set as a goal that the robot should perform random actions, or that it should roam. Once this is done, the planner should produce a plan to achieve this goal, and the Execution component should then execute this plan. The DesignOfExperiments component is responsible for designing effective experiments when the robot is in the experimentation state.

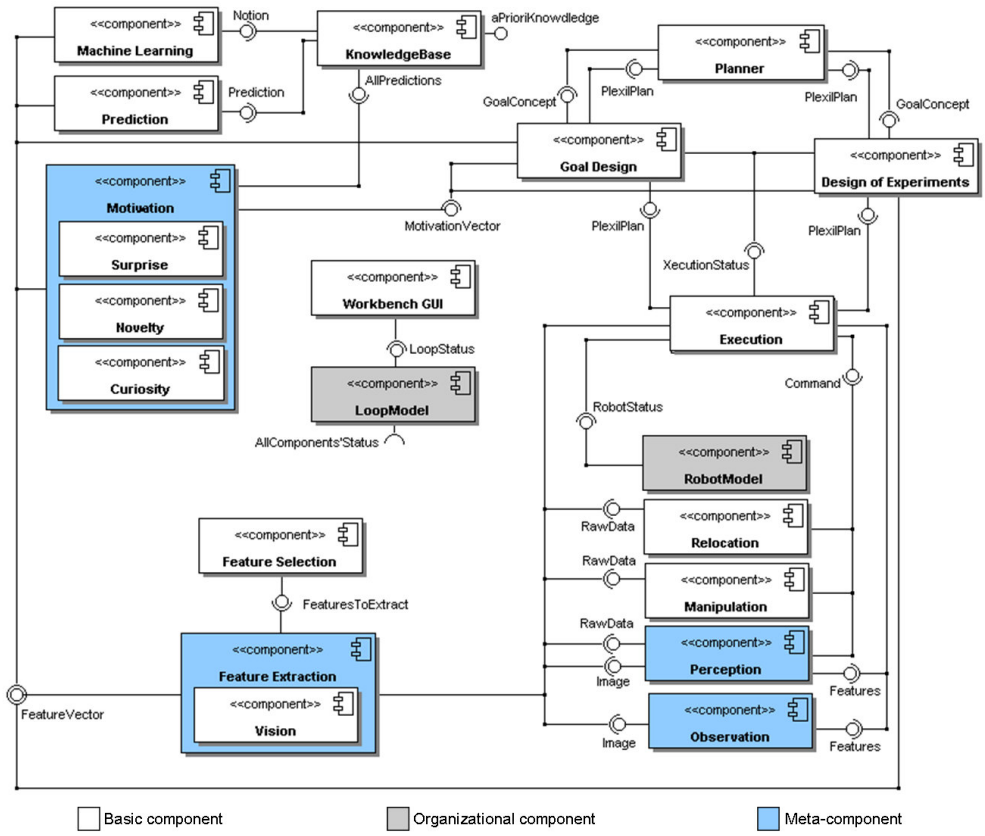

Figure 4.10: A component diagram showing the data flow between XPERSIF components.

Similarly, the concept is sent to the PLANNING component and then the EXECUTION component.

The robotic embodiment is itself represented by a group of components. As the embodiments are mobile manipulators, the components include a RELOCATION and a MANIPULATION component. In addition, a PERCEPTION component provides access to the embodiment's sensors. These components receive commands from the execution component and return monitoring information to it. A central point to query the embodiment's state is the ROBOTMODEL component. It is part of the organizational layer as it is responsible for starting those components which make up the robotic embodiment.

An overhead camera is often used within the cognitive loop both by the human researcher to tele-observe the experiment, and by the robot itself to provide ground truth. In the latter case, the view from the overhead camera is provided to the robot as a service (by the OBSERVATION component). Both this component and the PERCEPTION

component are displayed as meta-components in figure 4.10 as they use their own instances of the RobotFeatureExtraction meta-component. They are responsible for initializing the instances of this component and are the sole users of the interfaces.

The FeatureExtraction component takes as input the raw sensor data from the embodiment and the overhead camera and extracts meaningful features (by default objects and their poses). Should the FeatureSelection component specify additional features to be extracted, this may be carried out by either the FeatureExtraction component itself or the Vision component. The FeatureExtraction component generates a feature vector as the output of this process.

Components such as those of the Motivation meta-component and the MachineLearning component receive these feature vectors and use them to generate their own outputs (curiosity and surprise values, a prediction, a notion, etc). For the sake of simplicity, the diagram in figure 4.10 shows only the data flowing into the knowledge base, although it serves as a central point for all components to query for data at anytime.

4.5 Control flow within XPERSIF

This section presents a series of UML sequence diagrams which depict the operational specifications relating to each of the four states seen in figure 3.2 in addition to the initialization process of the loop. They show the flow of control within the loop and the responsibilities of the various components under its various states. It should be noted that this series of control flow diagrams serve to demonstrate one possible control flow – specifically, the one specified in the concrete movability experiment in chapter 3. While a concrete example is being presented the flexibility of the architecture in allowing a different flow of control can be seen. Another issue worth noting is the statelessness of the components themselves. The term 'states' here refers to the states of the cognitive loop as a whole. While components such as GoalDesign and DesignOfExperiments must be mindful of this information in order to perform their tasks, these states remain transparent to the rest of the components whose interfaces offer the same services regardless of the cognitive loop's state.

4.5.1 Loop Initialization sequence

The diagram in figure 4.11 provides a detailed view of the initialization process of the loop. This elaborates on the initialization process for a single component as described in section 4.2.

For the sake of simplicity, the details of the initialization process as presented in section 4.2 are omitted from the sequence diagram in figure 4.11. Each *startComponent* call

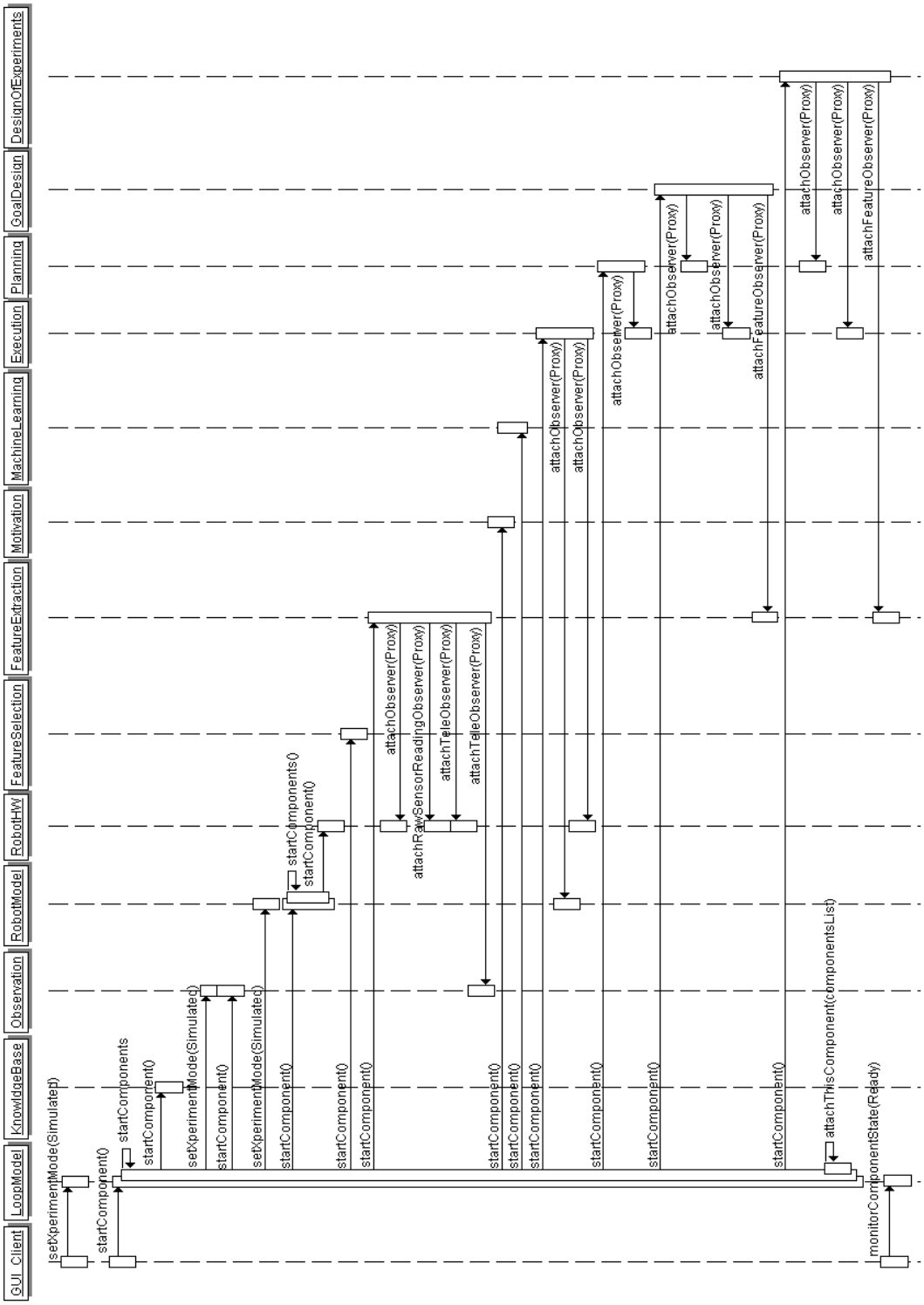

Figure 4.11: A UML sequence diagram showing the initialization of the experimental loop.

does proceed according to the description of the initialization process. The details of the robotic hardware's initialization process is also omitted for the sake of simplicity and a RobotHWComponents alias is used in place of the Relocation, Manipulation and Perception components. The initialization process for these components is specified in section 5.2.2 and sketched in figure 5.1.

4.5.2 State 0: The default loop state

This section details the flow of control in state zero (as seen in 4.12). As mentioned previously, the process mentioned here is for the specific concrete experiment depicted in the use case in Chapter 3.

The responsibility of maintaining and switching the state of the loop rests with the GoalDesign and DesignOfExperiments components as they are the ones responsible for choosing what tasks are being performed at any given moment. After LoopModel has started the components, the GoalDesign component checks that all components are ready (with a query to LoopModel). Once it receives this confirmation, it proceeds with the task of producing a goal concept for the planner (encapsulated within the Planning component). Once a plan is generated, it is returned to the GoalDesign component. This leads GoalDesign to request that the Motivation component start observing the feature vectors and compare them with the existing predictions within the Knowledgebase in order to generate the motivation vector (this request is accomplished through a call to the *continueComponent* operation). These vectors are transmitted to subscribers such as the DesignOfExperiments and FeatureExtraction components.

The plan execution proceeds as commands and operations are sent to the robot embodiment components and monitoring information is returned. While the execution is in progress, a steady stream of raw sensor data is sent to the FeatureExtraction component for use in producing the feature vectors. The plan continues to be executed until it times out or it is interrupted by GoalDesign. GoalDesign might halt execution of a plan if it detects a trigger which causes a state transition of the loop (in this case, from state zero to state one or two). As GoalDesign and DesignOfExperiments receive the Motivation vectors containing stimuli such as surprise detection or curiosity levels, they are able to orchestrate the components in order to proceed with the next states.

The inner workings of the Motivation and FeatureExtraction meta-components are detailed in sections 5.4.6 and 5.4.4.

In the given scenario, two possible states might follow the initial state of the loop. State one (surprise-validation) becomes the next state if the Motivation component's value

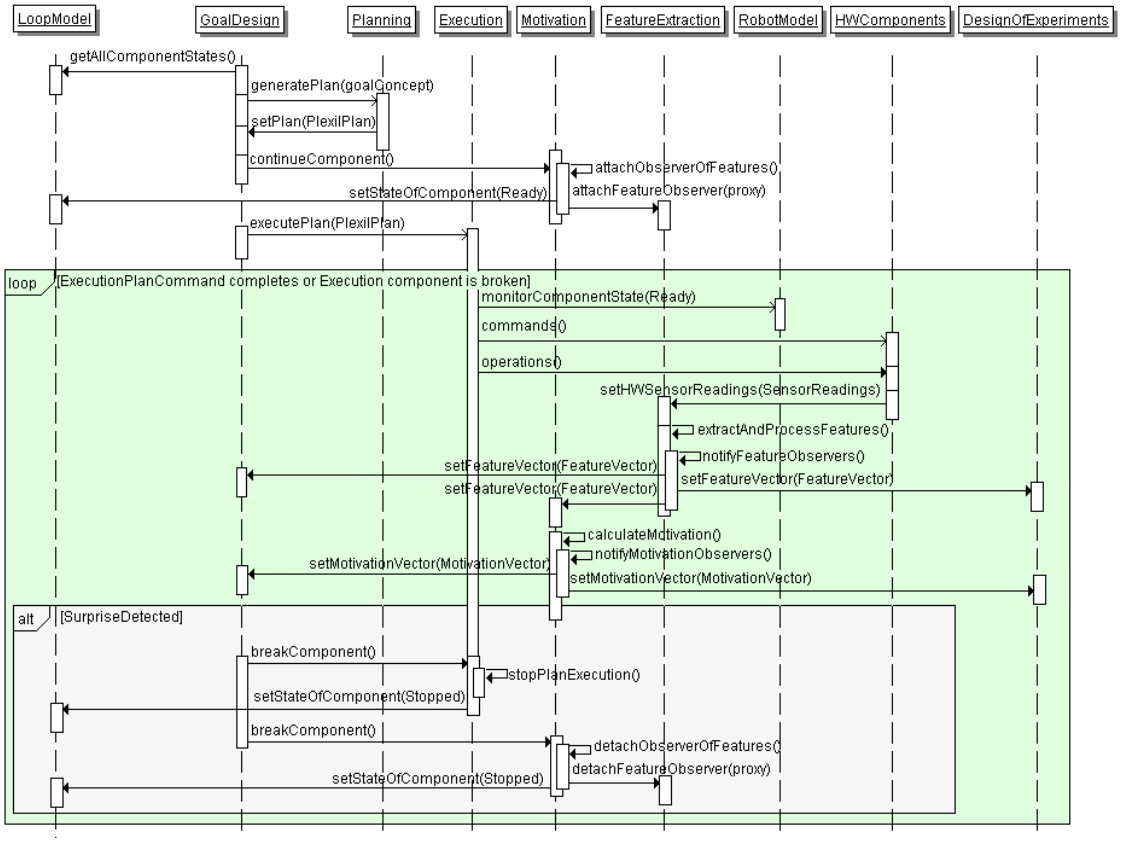

Figure 4.12: A UML sequence diagram showing an example of control flow in state zero of the XPERO experimental loop.

for surprise is above a given threshold. If any other form of motivation was sufficiently high, the loop transitions to state two (the experimentation state) directly and bypasses the surprise-validation state.

4.5.3 State 1: The surprise-validation state

In this state a series of attempts to reproduce the surprise trigger are carried out. The control flow of this state is seen in figure 4.13.

The GoalDesign component generates a new goal concept to validate the surprise which was detected. In accordance with the specifications of the movability experiment described in section 3.1, GoalDesign will first request that the Planning component attempt to generate a plan recreating the settings which led to the detection of surprise. If this plan's generation fails, observers of Planning are notified (which include GoalDesign) and the state transitions back to the default state. If it succeeds, GoalDesign then asks the planner to generate another plan which attempts to take the same action which led to the previous detection of surprise. Once again, if this plan

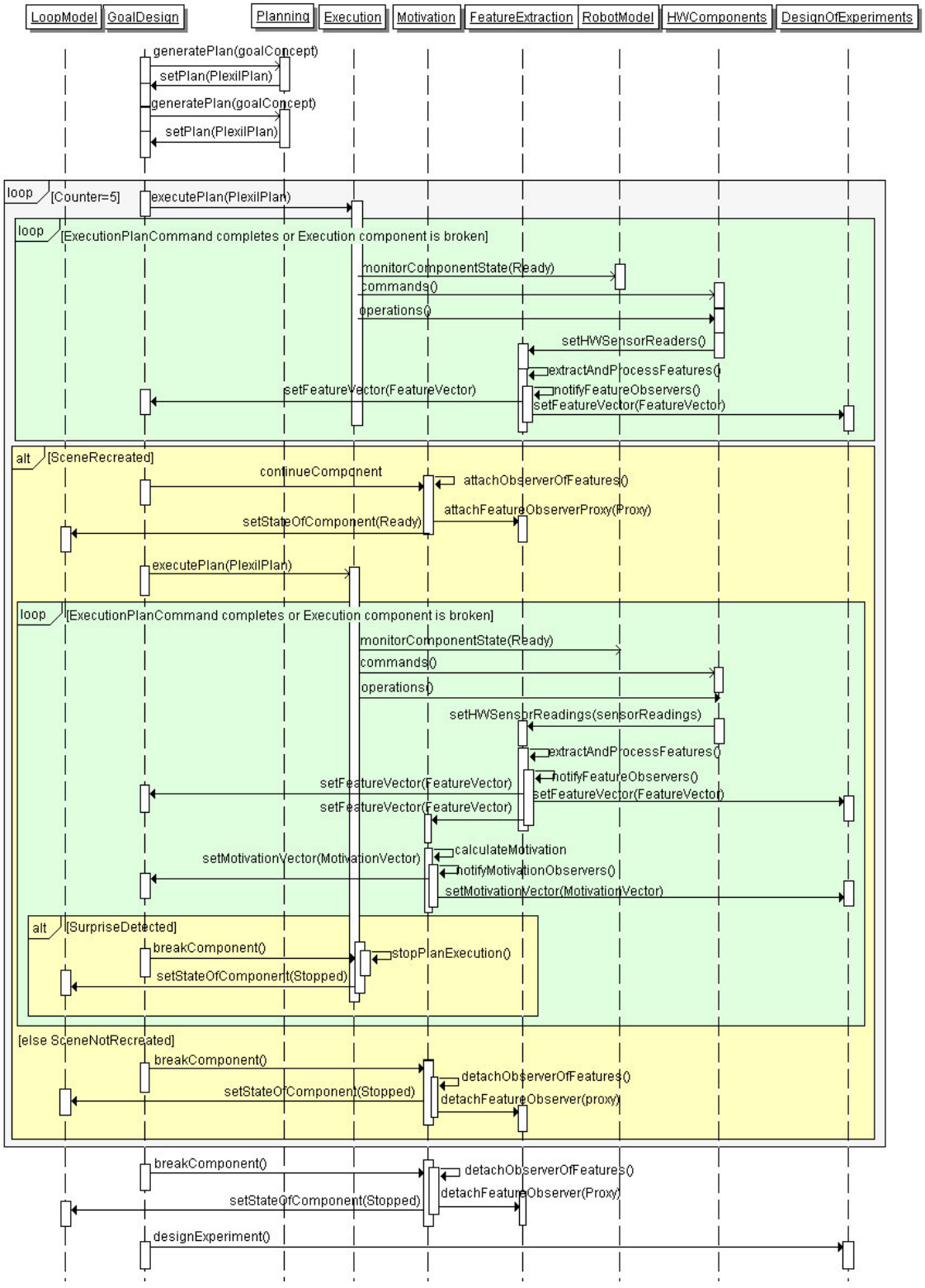

Figure 4.13: A UML sequence diagram showing an example of control flow in state one of the XPERO experimental loop.

generation process fails, the loop transitions to the default state.

GoalDesign asks Execution to execute the first plan. During the execution of this plan, no motivation vectors are generated since the component is still in the state: 'stopped'. Once the setting has been recreated, Execution notifies its observers. Should this first plan's execution fail all further attempts to validate surprise are abandoned. The loop then transitions back to the default state.

If the scene was recreated successfully, then GoalDesign continues the Motivation component (which completes when Motivation has once again subscribed to the feature vector). The control sequence continues as in state zero in terms of the execution of the plan and the generation of motivation vectors and hopefully the detection of surprise once more. If at any point during this execution, surprise is detected, not only is the Execution component braked as in state zero, but the Motivation component is also braked (as it is not needed in the scene-recreation phase).

This whole process of first recreating the settings and then executing the actions may be repeated for a pre-defined number of times (five times in figure 4.13). At the end of this process, the Motivation component must always be braked as new motivation vectors are no longer required. If surprise was triggered consistently, then the loop transitions to the experimentation state (state two) otherwise the loop transitions back to state zero. A call from GoalDesign to DesignOfExperiments enables this transition.

4.5.4 State 2: The experimentation state

Within the experimentation state of the loop, it is now the DesignOfExperiments component which designs an appropriate experiment. DesignOfExperiments is able to use the motivation vectors which it has so far been receiving for this task. The process of generating a plan, executing it and generating feature vectors from the observations is once again executed for a pre-defined number of times. Before the execution of the plan, DesignOfExperiments continues the MachineLearning component thus enabling it to receive the feature vectors which form the trace of the experiment. Once the repeated execution of the plan completes, the DesignOfExperiments component brakes the FeatureExtraction component as the feature vectors are no longer required. Once this has been accomplished, the loop transitions to state three: the learning state. The control flow is shown in figure 4.14.

4.5.5 State 3: The learning state

At this stage, the MachineLearning component has been buffering the feature vectors extracted from the observations obtained during the experimentation state. The DesignOfExperiments component signals it to start learning using the vectors. The

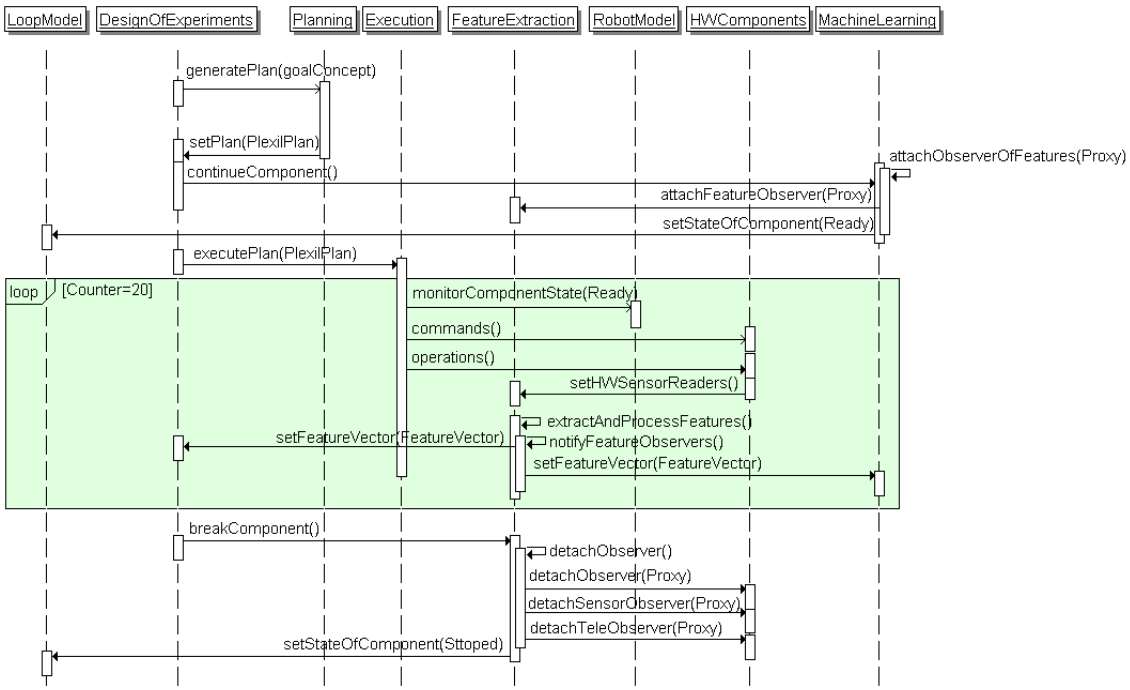

Figure 4.14: A UML sequence diagram showing an example of control flow in state two of the XPERO experimental loop.

inner workings of this component are not dealt with in this work. The loop is closed when this state transitions. If a new conclusion is drawn the whole loop may start over again in state zero. If the learning process fails, it may transition directly to state two in order to attempt to design another experiment and repeat the process once more. The control flow for this state is depicted in figure 4.15.

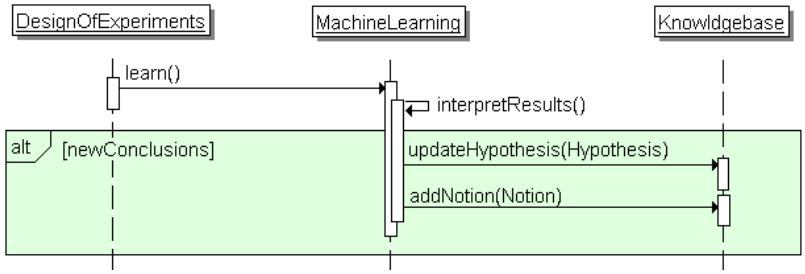

Figure 4.15: A UML sequence diagram showing an example of control flow in state three of the XPERO experimental loop.

4.5.6 Sumamry

Through this detailed control flow specification for a single concrete experiment, the effectiveness of the architecture in enabling it has been proven. Moreover, this specification reveals the flexibility of the architecture in enabling different control flows.

For example, the use of novelty detection or curiosity in place of surprise or in addition to it would necessitate changes in the implementation of the GoalDesign and DesignOfExperiments components only (which is acceptable as this is related to their inner workings and represents a change in the implementation). The interfaces of all the components provide services at a level of granularity that makes them stateless and remain unaffected by such changes. The mechanisms that the framework provides for communication (e.g. the observer pattern which enables notifications of all sorts) and for error handling (through the use of notifications and of ReturnCodes and CommandReturnCodes as seen in section 4.1.2) provide a solid framework for a robust architecture. While the design itself has these desired characteristics, only through the careful implementation of the design is a flexible and robust implementation of the architecture ensured. The implementation details are presented in Chapter 5.

Chapter 5

THE IMPLEMENTATION OF XPERSIF

This chapter details the systems' implementation. First the specifics of the implementation of the basic component model are presented. The details of the various components within the framework starting with the components of the organization layer are then presented. These are followed by those components which encapsulate the robotic hardware and the overhead camera. The software components which contain the application logic concerning the remainder of the loop are then covered. Next, the implementation focus on distributed simulation is presented.

The data types involved in any implementation provide key insights into the workings of a system. The process of specifying data types is part of the refining of an interface. The `XPERSIFDT.ice` file (Appendix B.2) contains all shared data types within the XPERSIF architecture. These are separate from the core data types found in the `XPERSIF.ice` file (Appendix B.1) which are directly related to the component model and framework.

The implementation details within this chapter vary in terms of the level of refinement. This is largely due to the continuing application development efforts (some of them being at more advanced stages than others) and the as yet unstable data and control flow of the loop. The hardware components have been available from the start and the refining of their interfaces was concentrated on enabling embodiment-independent services. The Vision component, for example, is at a far more advanced development stage where even data types may be specified. For other components, their interfaces are the result of the requirements analysis process which have been reached by thoroughly examining the application logic needed for the various use cases (see Chapter 3) as well as through discussions with the involved partners. These interfaces and the data types which they use will need to be continuously refined as the applications themselves and the continuously evolving flow within the loop mature. For this reason, the depth of the details of the implementation below varies widely.

5.1 The implementation of the component model

A number of the components below has been implemented as placeholders for the real applications provided by the partners within the project. These components provide working examples of implementations of various types of components within the architecture. They also serve to prove the viability and usability of the framework and

architecture. Details of the implementations of the functions of the IOperation and IOrganizationalOperation interfaces are presented here. A summary of these interfaces is presented below:

```
// The IOperation interface
ReturnCode startComponent();
ReturnCode brakeComponent();
ReturnCode continueComponent();
ReturnCode exitComponent();
ReturnCode getCommandTaskState(XPERSIF::CommandTaskID, out CommandTaskInfo);
ReturnCode getComponentState(out XPERSIF::ComponentState);
ReturnCode getRunningCommands(out XPERSIF::CommandTaskIDlist);

// The IOrganizationalOperation interface
ReturnCode registerComponent(XPERSIF::ComponentID componentID);
ReturnCode getAllComponentStates(out ComponentStateInformationList stateList);
ReturnCode getStateOfComponent(XPERSIF::ComponentID componentID, out XPERSIF::ComponentState
    componentState);
ReturnCode setStateOfComponent(XPERSIF::ComponentID componentID, ComponentState componentState);
CommandReturnCode monitorComponentState(XPERSIF::ComponentState componentState);
```

The *startComponent* operation is called by an organizational component (or by the client in the case of LoopModel) to start any given component. The component first obtains the proxy of the organizational component with which it must register itself. This is done by calling the *registerComponent* operation of the organizational component which manages it. Upon successfully registering, a component proceeds with its initialization. Some components may not need to perform any sort of initialization process and will thus proceed immediately to notify their organizational component that they are 'Initialized'.

In the case of organizational components and meta-components, this initialization might include the starting up of other components or subcomponents. As this process may take time, it is carried out by a thread spawned from within this function. The *startComponent* function then returns a ReturnCode to its caller signaling the success or failure of the process so far. At this point, these components' state is still 'Registered'.

Upon completing the initialization process within the thread, a *notifyOnStart* helper function within the implementation file for the component is called. This function notifies the relevant organizational component of the new component state 'Initialized' through the operation *setStateOfComponent*.

For all components which need to observe a component or a component's output (e.g. feature vectors provided by the FeatureExtraction component), a thread is spawned to subscribe the component to any given set of components and to any given vectors. Upon completion, this thread notifies the relevant organizational component that the component's state is 'Ready'.

In all cases where notification of a component's state is changed (as described above), this is accomplished through a call to a helper function *setMyComponentState*. This ensures that the current copy of the component's state is also updated safely (using mutexes), and that the correct organizational component is consistently called. It also ensures consistency between the local copy of the component state and the state that the organizational component has. The implementation of this function as a local one allows it to be called from any and all local functions and threads within the component.

The thread responsible for subscribing to observe other components and their output must poll for the status of these components with the LoopModel component. If the state is anything but 'Unknown', this subscription using the IObserevrOf interfaces within the component Slice definitions is made.

It should be stressed that all processes which do not complete immediately (such as commands and other processes involving polling for something – such as a component's state) must be implemented as threads to ensure that no blocking and deadlocks occur. The use of mutexes and the accessing of variables through getter and setter functions is also recommended.

The *brakeComponent* and *continueComponent* operations are used to signal components to subscribe and unsubscribe from receiving vectors (but not notifications from components that are observed). The *brakeComponent* function starts by calling the helper function *getProxies*. This ensures that the component has the correct proxies for the components it will detach from and that it has its own proxy. This is used to recreate the callback proxy which was sent to subscribe in the first place. This callback is not available locally as it was performed within a thread. Once the proxies are obtained, the callback is recreated and the publishers of the output are called. Once the detaching of the component returns, the component state is changed to 'Stopped' (using the helper function *setMyComponentState* as explained above).

Similarly, the *continueComponent* operation gets the proxies of the components it needs before recreating the callbacks and sending them as argument to the relevant components in order to subscribe to their output. The function completes by setting the component state to 'Ready'.

The *exitComponent* operation is called by the organizational component which started the component before the application is shut down. This function unsubscribes the component as in the *brakeComponent* function. This time, this also includes the notifications of command execution from components it observes. Before finishing, the function calls a helper function *setStopFlag* which sets a flag to signal that the

component is exiting. This flag is regularly polled for by threads which use a **while** loop and ensures that they cleanly exit. Naturally, the notification mechanisms described in section 4.1.3 is adhered to and in the case a command exists because of the flag being set to 'true', the CommandReturnCode within the CommandTaskInfo structure would specify that the command was not completed and the description within the data type would be filled accordingly. The consistent use of these mechanisms is vital for error handling and provides a robust implementation.

The *getCommandTaskState* operation allows any component to query the status of a specific command by sending the CommandTaskId of the command. Each component maintains a list of this data structure – one for each command which it has received. This list is checked for the matching data structure and used as part of the CommandTaskInfo structure and then returned. If the entry was not found, this is specified within the ReturnCode. The *getComponentState* operation allows components to query each other's state. Finally, the *getRunningComands* operation returns a list of all the component's running commands (using the same list specified above). Similarly, the *getAllCompoentStates* operation would fetch the list and return a ComponentStateInformationList.

In addition to the commands and operations mentioned above, the implementation of the observer pattern for subscribing and unsubscribing is also shared. As mentioned in section 4.1.3, the process of subscribing and unsubscribing to observe components and their output is enabled through the attaching and detaching of observers. The implementation of the subscription process by the observer has been explained above. The implementation of the relevant *attach* and *detach* operations are explained here.

The *attachObserver* operation simply gets the list of observer proxies stored locally (through a helper function using a mutex) and adds the proxy of the observer to it before finally setting the list (again through a helper function). The *detachObserver* operation similarly gets the list, searches for the given proxy which it should detach and once it has been found, erases it. A flag is set to signal whether the given proxy was found in the list. If not, this is reflected in the ReturnCode. Finally, the list (without the proxy) is set once again. The same implementation is followed for subscribing and unsubscribing observers of output such as feature vectors, etc.

5.2 The organizational components

Generally, an organizational layer is responsible for managing hierarchies. This allows the overall architecture to remain flexible by centralizing the control. This further encourages the loose coupling of components in addition to providing a central point to query and parameterize components within the hierarchies.

Components within this layer are based on the component model presented in section 4.1 but include functionality which allows them to monitor any given set of components. Detailed information about the functionality and structure of these components is presented in section 4.2. The specifics of the implementation are presented here. The organizational components within the architecture are shown in figure 4.10.

5.2.1 The LoopModel component

This component provides a single point of entry to the whole experimental loop. It is used for parameterization of the various components and their initialization and termination. This initialization sequence is covered in section 4.5.1. The LoopModel component is the sole component within the LoopControl application. It is started (and closed) with a call from the client which for now takes the place of the workbench GUI.

Component name: LoopModel
Component type: Organizational
Application: LoopControl
Slice definition: `LoopModel.ice` (in section B.4)
Organized by: Client
Organizes: GoalDesign, DesignOfExperiments, Planning, Execution, RobotModel, Observation, FeatureSelection, FeatureExtraction, Motivation, Prediction, MachineLearning
Observes: all components it organizes
Subscribes to the outputs: none
Publishes the outputs: none
Status: Complete

The component implements those operations within the IOperation interface as well as those within the IOrganizationalOperation interface. Other organizational components must also implement these functions.

5.2.2 The RobotModel component

This component is responsible for the initialization and configuration of the robotic hardware components and is therefore considered as an organizational component. This process is depicted in figure 5.1. For the sake of simplicity, the details of the initialization process of the FeatureExtraction component are presented fully in section 5.4.4. Additionally, it provides a single point for components, such as Execution for example, to query the state of the robotic hardware.

Component name: RobotModel
Component type: Organizational

Application: RobotControl
Slice definition: `RobotModel.ice` (in section B.5)
Organized by: LoopModel
Organizes: Manipulation, Relocation, Perception
Observes: all components it organizes
Subscribes to the outputs: none
Publishes the outputs: none
Status: Complete

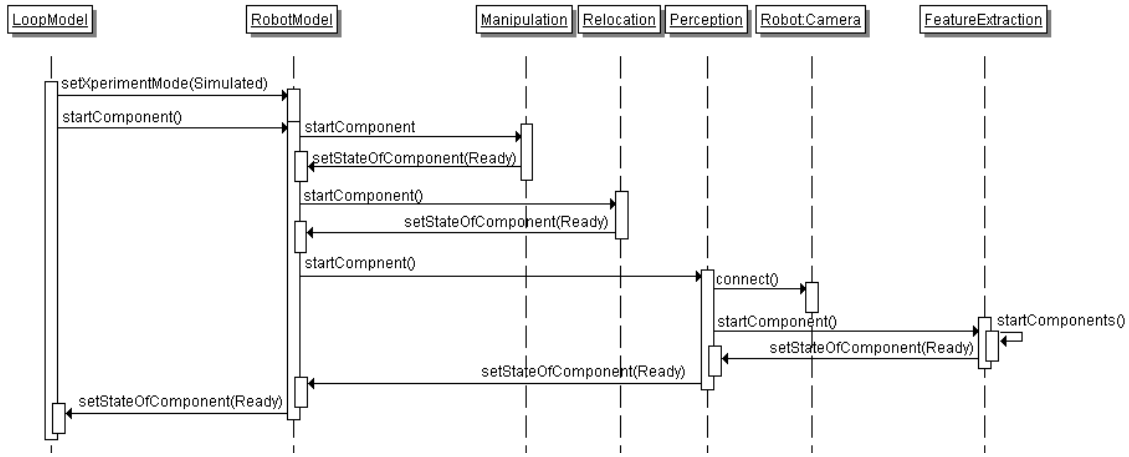

Figure 5.1: A UML sequence diagram showing the initialization of the robotic hardware components by the RobotModel component.

The RobotControl application includes the implementation of the hardware components which RobotModel is responsible for, namely: Relocation, Manipulation and Perception. The implementation of these components is presented in the following section.

5.3 The hardware components

This section deals with the implementation of the components which encapsulate the functionality of the hardware involved in the loop – namely the robotic embodiment and the overhead camera. The emphasis was on providing interfaces that are as abstract as necessary in order to ensure flexibility in using multiple embodiments. The resulting interfaces were developed by pursuing a top-down approach from the use cases for the Execution component (found in Appendix D).

5.3.1 Hardware abstraction mechanism

As portability across the embodiments is a necessity, a method to abstract away the embodiments is needed. The process of specifying interfaces for the embodiment is

simplified by the assumption that the robots are mobile manipulators. The task is still a complicated one given the variety in both mobile platforms and in manipulators. This assumption also simplifies the task of the EXECUTION component as all plans may involve relocation and manipulation tasks. At some level, the commands must become embodiment-specific. This level is found just below the component implementation classes within a *skills* class.

For each embodiment, there exists a skills class which implements a basic set of functions used by the robot components' implementation and *status* classes to send commands to, and receive data from, the specific embodiment.

For example, the KheperaSkills class allows the relocation component to read the velocity of the drive through a function call which translates the operation into a Khepera API call and executes it through the relevant client (through a serial client if the physical robot is used or through an Ice client if the experiment is simulated). Once the result is received, the value must be transformed into SI units which are drive-independent - e.g. linear velocity in the x and y directions and angular velocity. In this way, the implementation classes of the components are embodiment-independent. All information which they send and receive is provided in SI units and in the case of the relocation component, drive-independent.

While the implementation class is always embodiment independent, it may occasionally be necessary for the status classes to include embodiment-specific implementations. This may be seen clearly when dealing with manipulators where the methodology used to reach a pose differs greatly depending on its degrees of freedom.

The XPERSIF framework itself simplifies the abstraction process through the use of commands and the notification mechanism as it allows other components to work independently. For example, the command *grip* found within the MANIPULATION component's interface does not require the EXECUTION component to specify a time, or a distance, or a value which is dependent on the embodiment. The control of the manipulator in order to execute the action is all carried out behind the interface, where the embodiment is no longer abstract but concrete. The notification of the status of command execution is the only information needed.

5.3.2 Robot manipulation

The implementation of the MANIPULATION component is found in the **ManipulationI** class.

Component name: MANIPULATION

Component type: Basic
Application: RobotControl
Slice definition: `Manipulation.ice` (in section B.6)
Organized by: RobotModel
Organizes: none
Observes: none
Subscribes to the outputs: none
Publishes the outputs: ManipulationRawSensorReadings
Status: Complete

As in the example component found in section 4.1, it implements the set of functions declared in the IOperation interface. In addition, the class implements the functionality specific to manipulators. The following functions from the interface have been implemented:

```
XPERSIF::CommandReturnCode grip();
XPERSIF::CommandReturnCode ungrip();
XPERSIF::CommandReturnCode moveTooltipToPose(XPERSIFDT::Pose6D pose);
XPERSIF::CommandReturnCode prepareToRelocate();
XPERSIF::ReturnCode getTooltipPose(out XPERSIFDT::Pose6D manipulatorPose);
XPERSIF::ReturnCode getGripperPose(out double distance);
XPERSIF::ReturnCode isObjectPresent(out bool objectPresent);
```

The Slice definitions for all components include documentation on preconditions and postconditions as well as an explanation of the functionality of the command and its parameters.

Functions from the IManipulationSubject interface are also implemented and allow components wishing to receive notification of command execution status to subscribe and unsubscribe to them.

```
// Functions from Manipulation::IManipulationSubject
XPERSIF::ReturnCode attachObserver(IObserverOfManipulation *);
XPERSIF::ReturnCode detachObserver(IObserverOfManipulation *);
```

For commands, *status* classes run in threads which monitor the status of the necessary tasks and call the *notifyObservers* function within the **ManipulationI** class when a task is completed (whether the task completed successfully or unsuccessfully). This function is called by the various status classes within the Manipulation component to notify all observers who have registered to receive notifications from the component. The same function is implemented in each of the components for use by its status classes.

The **ManipulationGripStatus** class polls the distance between the grippers and calls the *notifyObservers* function once the distance stabilizes for a period of time.

The **ManipulationUngripStatus** class also polls for the distance between grippers but only notifies once the grippers are at their maximum distance from each other, meaning that the gripper is completely open.

The *moveTooltipToPose* command moves the tool-tip to the given pose which is specified in robot-centric coordinates and refers to the pose which will be achieved when the gripper closes. The *prepareToRelocate* command moves the manipulator to a pose that is safe during relocation. As this pose is embodiment-specific, no arguments are passed to this command. The implementation within the status class implementing this command must specify this pose for each embodiment.

The operations within this component include *getTooltipPose* which delivers the pose of the tool-tip in robot-centric coordinates, *getGripperPose* which delivers the distance between the inner surfaces of the grippers in meters, and *isObjectPresent* which delivers the presence (or lack thereof) of an object between the grippers.

Finally, the **ManipulationI** class also implements the interface IManipulationRawSensorReadingsSubject which allows components (such as FeatureExtraction) to receive raw sensor data from the component. This interface works in the same way as the IManipulationSubject interface. The following functions are used to subscribe and unsubscribe to the component's sensor data.

```
XPERSIF::ReturnCode attachRawSensorReadingsObserver(IObserverOfManipulationRawSensorReadings *);
XPERSIF::ReturnCode detachRawSensorReadingsObserver(IObserverOfManipulationRawSensorReadings *);
```

The sensor data is updated periodically through a call from a specialized class (**ManipulationRawSensorReader**) to the skills class. The skills class then executes a series of API calls to retrieve the sensor readings. The same thread then calls the function *notifyObserversOfRawSensorReadings* within `ManipulationI.cpp` which proceeds with the notification process. This mechanism allows the skills class to act as a blackboard for all raw sensor data. These readings within the skills class are also used by status classes executing the commands. The **ManipulationRawSensorReader** is started in the Manipulation component's *startComponent* function. The same mechanism is used by both the Relocation and the Perception component to provide raw sensor readings to subscribers.

5.3.3 Robot relocation

The RELOCATION component encapsulates the functionality necessary for robot locomotion and motion control. The RELOCATION component is implemented in the **RelocationI** class which implements the RelocationComponent interface.

Component name: RELOCATION
Component type: Basic
Application: RobotControl
Slice definition: `Relocation.ice` (in section B.7)
Organized by: ROBOTMODEL
Organizes: none
Observes: none
Subscribes to the outputs: none
Publishes the outputs: RelocationRawSensorReadings
Status: Complete

It is often necessary for a robot to manoeuver from an initial pose to a goal pose. Mobile platforms that allow a robot to move come in many forms, such as differential drives and omni-drives. The choice of velocities with which to move in order to achieve this goal pose is an inverse kinematics problem. Figure E.1 depicts this problem.

Depending on the drive a robot uses, the method of specifying velocities differs. Some platforms require two velocities to be specified (as in the case of a differential drive). Others may require three velocities to be calculated. The relocation component provides transparency between the embodiments by using poses within its interface and handling velocities in their x, y and ω components. Thus, all mobile platforms are treated in the same way and the same relocation commands can be sent to them without any need for changes to be made.

A controller has been implemented to provide the motion control necessary for the functionality specified within the interface of this component. The implementation is found within its own class to facilitate its re-usability by the various relocation-status classes. It is supported by helper functions found in the RELOCATION component's implementation class – **RelocationI**. These functions implement such calculations as the conversion to and from differential-drive velocities and obtaining the distance between two poses.

The controller which has been implemented uses the approach defined in [1] where the authors apply the Lyapunov theory to obtain effective closed-loop control laws for unicycle-like vehicles. The novelty of the authors' approach comes in the choice of the

system of kinematic equations (system state equations) which facilitate designing appropriate closed-loop control laws for the vehicle manoeuvering [1]. A detailed description of the inverse kinematics problem and the implemented controller can be found in Appendix E.

A summary of the commands and operations within the IRelocation interface is provided here:

```
XPERSIF::CommandReturnCode moveAbs(XPERSIFDT::Pose pose);
XPERSIF::CommandReturnCode moveRel(XPERSIFDT::Pose deltaPose);
XPERSIF::CommandReturnCode followPathAbs(XPERSIFDT::Path path);
XPERSIF::CommandReturnCode followPathRel(XPERSIFDT::Path path);
XPERSIF::CommandReturnCode moveAbsInReverse(XPERSIFDT::Pose pose);
XPERSIF::CommandReturnCode moveRelInReverse(XPERSIFDT::Pose deltaPose);
XPERSIF::CommandReturnCode moveForward();
XPERSIF::CommandReturnCode moveForwardAtSpeed(double speed);
XPERSIF::CommandReturnCode moveBackward();
XPERSIF::CommandReturnCode moveBackwardAtSpeed(double speed);
XPERSIF::CommandReturnCode rotateByAngle(double angle);
XPERSIF::CommandReturnCode rotate();
XPERSIF::CommandReturnCode stopDriving();
XPERSIF::ReturnCode setMaximumRobotVelocities(XPERSIFDT::Velocity velocity);
XPERSIF::ReturnCode setAccuracy(double accuracyPosition, double accuracyYaw);
XPERSIF::ReturnCode getCurrentSpeed(out XPERSIFDT::Velocity velocity);
XPERSIF::ReturnCode getCurrentPose(out XPERSIFDT::Pose currentPose);
```

The **MoveAbsStatus** class is the status class which monitors the progress of the *MoveAbs* command. This command is used to move to a pose given in absolute coordinates. The status class creates an instance of the controller and calls the *driveTo* function with a Pose data type (a tuple: x,y in meters and yaw in radians) as argument. When the given pose is achieved, the class will call *notifyObservers* to perform the notification of all observers of RELOCATION.

Similarly, the **RelocationMoveRelStatus** class monitors the *moveRel* command which allows the robot to move to a pose given in egocentric coordinates. It first calls the helper function *convertToAbsPose* (within `RelocationI.cpp`) to obtain an absolute position to call the controller with.

The *FollowPathAbs* function takes a series of tuples (poses) which comprise a Path. Its status class **RelocationFollowPathPathAbsStatus** calls the controller with a given pose, waits until the pose is reached and then proceeds to call the controller with the next pose. Notification is performed once the entire path has been traversed.

The **RelocationFollowPathRelStatus** class, in similar fashion to the **MoveRelStatus** class, will convert a pose to absolute coordinates, call the controller

and upon reaching it, will then convert the following pose and continue until the given poses have been achieved. Once again, notification is called when the final pose has been achieved.

A more primitive set of commands have also been implemented to allow the embodiments to relocate without the use of the controller. These include the commands: *moveForwardAtSpeed*, *moveBackwardAtSpeed*, *moveForward*, *moveBackward*, *rotate* and *rotateByAngle*. The *stopDriving* command stops relocation by sending velocities of zero. It polls for the velocity of the robot and once this reaches zero, the notification is performed.

The *getCurrentPose* operation returns the pose of the robot obtained through the OBSERVATION component as this value is far more accurate than the dead-reckoning method which uses encoder readings. The *getCurrentSpeed* operation returns the speed of the robot in its linear and angular components. With the *setAccuracy* operation, it is possible to set the accuracy values for position and orientation which the controller uses to determine if it has in fact reached the given pose.

The RELOCATION component implements the IRelocationRawSensorReadingsSubject allowing components to subscribe to the raw sensor readings available from this component.

5.3.4 Robot perception

The PERCEPTION component provides the robot's perception of its environment. It does this by making available raw sensor readings that do not belong to the MANIPULATION or RELOCATION components (e.g. images, IR sensor readings, etc). In addition, it also provides features which may be necessary for the execution of a given task (e.g. distance between the robot and a given object). The implementation of the PERCEPTION component is found in the **PerceptionI** class. The class implements the IPerceptionComponent interface.

Component name: PERCEPTION
Component type: Meta-Component
Application: RobotControl
Slice definition: `Perception.ice` (in section B.8)
Organized by: ROBOTMODEL
Organizes: ROBOTFEATUREEXTRACTION, CAMERA
Observes: all components it organizes
Subscribes to the outputs: none
Publishes the outputs: Perception raw sensor readings, robot camera image(s)

Status: Missing the integration of the RobotFeatureExtraction component and the Camera subcomponent

The Perception component contains no commands within its interface – only operations. These include operations to get features (e.g. distances, poses, angles) in addition to operations for attaching observers (although, as no commands are included in the interface, no notifications would be received), and a variety of camera-related operations. It also allows those wishing to tele-observe the robot camera's view to attach as tele-observers. The basic operations provided within the IPerception interface are summarized below:

```
XPERSIF::ReturnCode getObjectsInView(out XPERSIFDT::DetectedObjectList detectedObjectList);
XPERSIF::ReturnCode getObjectPose(XPERSIFDT::ModelIDs objectLabelID, out XPERSIFDT::Pose pose);
XPERSIF::ReturnCode isObjectInView(XPERSIFDT::ModelIDs objectLabelID, out bool inView);
XPERSIF::ReturnCode getObjectArea(XPERSIFDT::ModelIDs objectLabelID, out double area);
XPERSIF::ReturnCode getObjectCenterOnScreen(XPERSIFDT::ModelIDs objectLabelID, out XPERSIFDT::Point2D
    centerOnScreen);
XPERSIF::ReturnCode getTypeOfProcessing(out XPERSIFDT::CameraTypeOfProcessing typeOfProcessing);
XPERSIF::ReturnCode getCameras(out XPERSIFDT::CameraList cameraList);
XPERSIF::ReturnCode getDistanceBetweenObjects(XPERSIFDT::ModelIDs objectLabelID1, XPERSIFDT::ModelIDs
    objectLabelID2, out double distance);
XPERSIF::ReturnCode getAngleBetweenObjects(XPERSIFDT::ModelIDs objectLabelID1, XPERSIFDT::ModelIDs
    objectLabelID2, out double angle);
XPERSIF::ReturnCode getIR(out XPERSIFDT::SensorReadings IRreadings);
XPERSIF::ReturnCode getBumper(out XPERSIFDT::SensorReadings bumpSensorReadings);
XPERSIF::ReturnCode getTime(out double time);
```

The Perception component is a meta-component in that it contains its own instance of the RobotFeatureExtraction meta-component (its operations may only be used through the Perception interface). This is necessary as the architecture aims to clearly separate the functionality of logging and execution. This ensures that the delivery of time-sensitive information which may be vital to the successful execution of a task is not compromised by burdening the component with additional calculations (which may be necessary for creating feature vectors for the logging process). The RobotFeatureExtraction component is introduced here but is detailed in section 5.4.4. The instance used by Perception extracts only those features which the component is required to produce for other components. In addition to its own instance of the RobotFeatureExtraction component, it also uses the Camera subcomponent which encapsulates the functionality of the camera sensor. This subcomponent is detailed in the following section. The specifics of the interaction between Perception and RobotFeatureExtraction are shown in figure 5.2.

Two functions allow anyone wishing to tele-observe the robot camera's view to subscribe and unsubscribe to it. These are *attachImageSimObserver* and *detachImageSimObserver*

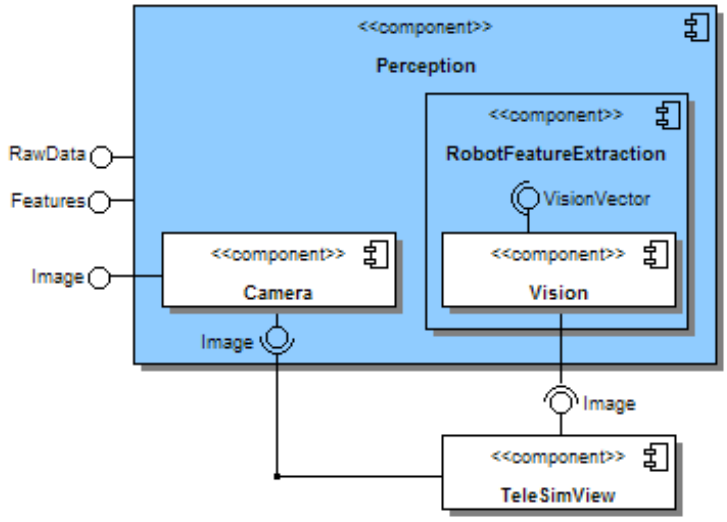

Figure 5.2: The data flow for the Perception component.

in the simulated setting (or *attachImageObserver* and *detachImageObserver* in the physical setting). A call to subscribe would in fact lead to the PERCEPTION component forwarding this request to the CAMERA subcomponent. As the image-provider, CAMERA would be the one multicasting the image to subscribers (in both the real and simulated settings). Those wishing to tele-observe use a client to do so. In the simulated setting, the TeleSimView client is the one to subscribe with the PERCEPTION component and is the one receiving the image and rendering it. In the physical setting, a similar client would follow the same process. The details involved in distributed simulation are covered in section 5.6. The implementation of the client for the physical setting is beyond the scope of this work.

The PERCEPTION component receives operations requesting features. PERCEPTION simply relays this request to its instance of ROBOTFEATUREEXTRACTION. ROBOTFEATUREEXTRACTION extracts the feature and returns it to PERCEPTION which then returns the value to the caller.

Objects and their poses are often required for these features to be calculated. The ROBOTFEATUREEXTRACTION component is responsible for these calculations and uses its own instance of the VISION component to supply this vision vector containing the detected objects and their poses. A viewing client is used by VISION to receive the images. As this work deals with the simulated setting, a TeleSimView client is shown in figure 5.2.

The CAMERA subcomponent sends images to this TeleSimView client. The VISION component requests a frame from it. The latest frame is delivered. VISION processes the

image, extracting all detected objects and their poses. These basic features are then used by the RobotFeatureExtraction component to calculate the requested features. If the feature requested may be extracted by the Vision component itself (e.g. area of an object on screen) then this is carried out by that component and returned to Perception (but always through the RobotFeatureExtraction component).

The same mechanism (described in 5.3.2) for updating sensor readings within the skills class is used by the Perception component through its own class. The initialization process for the Perception component, the Camera subcomponent and the instance of the RobotFeatureExtraction component may be seen in figure 5.1 as part of the overall robot initialization sequence.

5.3.5 Camera subcomponent

The Camera subcomponent serves as the image-provider. Other components wishing to use images from a camera will receive them from a Camera subcomponent. The subcomponent may not be accessed directly. Rather, it is accessed through either the Perception component (if its the robot camera which is needed) or through the Observation component (in the case of the overhead camera).

Component name: Camera
Component type: Subcomponent
Application: RobotControl and/or OverheadCameraControl
Slice definition: `Camera.ice` (in section B.10)
Organized by: Perception and/or Observation
Organizes: none
Observes: none
Subscribes to the outputs: XPERSim
Publishes the outputs: images from the robot camera's view or the overhead camera's view
Status: Complete for the simulated setting

The term *subcomponent* is used as the camera is not bound to the component model shown in figure 4.1. The interface provides the functionality to set camera parameters in addition to two functions which are used to start and stop the subcomponent. Additionally, it contains the *attach* and *detach* functions which are used to add subscribers and remove them. The basic operations within the ICamera interface are listed here for reference:

```
XPERSIF::ReturnCode getCameraInfo(out XPERSIFDT::CameraInfo cameraInfo);
XPERSIF::ReturnCode getCameraPosition(out XPERSIFDT::CameraPosition cameraPosition);
```

```
XPERSIF::ReturnCode setCameraId(int cameraID);
XPERSIF::ReturnCode setIntrinsicParameters(XPERSIFDT::Matrix intrinsicMatrix);
XPERSIF::ReturnCode setExtrinsicParameters(XPERSIFDT::Matrix extrinsicMatrix);
XPERSIF::ReturnCode setDistortionParameters(XPERSIFDT::Row distortions);
XPERSIF::ReturnCode setImageFormat(XPERSIFDT::ImageFormat imageFormat);
XPERSIF::ReturnCode setImageSize(XPERSIFDT::ImageSize imageSize);
XPERSIF::ReturnCode setCameraPosition(XPERSIFDT::CameraPosition cameraPosition);
XPERSIF::ReturnCode connect();
XPERSIF::ReturnCode disconnect();
```

The CameraInfo data type contains all the necessary parameters for the VISION component to provide its services.

5.3.6 Overhead Camera

The OBSERVATION component encapsulates the functionality of the overhead camera. It is not only used to enable human researchers to view the experiment but is also used as a service to the robot by delivering ground truth (e.g. providing robot localization for the RELOCATION component).

Component name: OBSERVATION
Component type: Meta-Component
Application: OverheadCameraControl
Slice definition: `Observation.ice` (in section B.9)
Organized by: LOOPMODEL
Organizes: ROBOTFEATUREEXTRACTION, CAMERA
Observes: all components it organizes
Subscribes to the outputs: none
Publishes the outputs: robot camera image(s)
Status: Complete

The same camera-related operations which are present in the PERCEPTION component are included here. This includes the functions to subscribe for tele-observation. OBSERVATION also instantiates a dedicated ROBOTFEATUREEXTRACTION meta-component for its use. The whole process is identical to the one described in section 5.3.4. The specifics of the interaction between OBSERVATION and ROBOTFEATUREEXTRACTION are shown in figure 5.3.

It is worth noting that the major difference between the PERCEPTION and OBSERVATION components is the lack of raw sensor data provided by the OBSERVATION component. Naturally, the view from the robot camera in PERCEPTION is much more limited than that available from OBSERVATION. Below is a list of operations available in the IObservation interface:

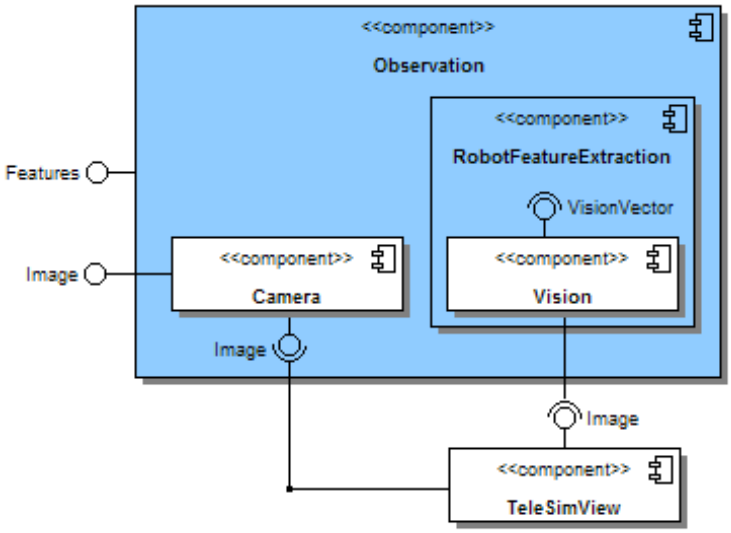

Figure 5.3: The data flow for the Observation component.

```
void setXperimentMode(XPERSIFDT::XperimentMode xperimentMode);
XPERSIF::ReturnCode getObjectsInView(out XPERSIFDT::ModelIDsList detectedObjectList);
XPERSIF::ReturnCode getObjectPose(XPERSIFDT::ModelIDs objectLabelID, out XPERSIFDT::Pose pose);
XPERSIF::ReturnCode getRobotPose(out XPERSIFDT::Pose robotPose);
XPERSIF::ReturnCode isObjectInView(XPERSIFDT::ModelIDs objectLabelID, out bool inView);
XPERSIF::ReturnCode getTypeOfProcessing(out XPERSIFDT::CameraTypeOfProcessing typeOfProcessing);
XPERSIF::ReturnCode getCameras(out XPERSIFDT::CameraList cameraList);
XPERSIF::ReturnCode getDistanceBetweenObjects(XPERSIFDT::ModelIDs objectLabelID1, XPERSIFDT::ModelIDs
    objectLabelID2,out double distance);
XPERSIF::ReturnCode getAngleBetweenObjects(XPERSIFDT::ModelIDs objectLabelID1, XPERSIFDT::ModelIDs
    objectLabelID2, out double angle);
XPERSIF::ReturnCode getTime(out double time);
```

5.4 The software components

This section deals with the implementation of the software components within the loop. The emphasis was on providing interfaces that allow the architecture itself to remain flexible and scalable.

5.4.1 Design components

There exist two components which are responsible for providing high-level goal concepts to a planner. These are the GoalDesign and DesignOfExperiments components. These high-level concepts may be the goal state for which the planner needs to generate a plan. The GoalDesign component is responsible for generating such concepts in the default and surprise-validation states of the loop while the DesignOfExperiments component takes the responsibility for this in the experimentation state.

Component name: GoalDesign

Component type: Basic
Application: GoalDesign
Slice definition: `GoalDesign.ice` (in section B.11)
Organized by: LoopModel
Organizes: none
Observes: Planning, Execution
Subscribes to the outputs: Feature vectors, Motivation vectors
Publishes the outputs:
Status: Complete placeholder (Implemented for state zero of the movability experiment)

Component name: DesignOfExperiments
Component type: Basic
Application: not implemented
Slice definition: `DesignOfExperiments.ice` (in section B.12)
Organized by: LoopModel
Organizes: none
Observes: Planning, Execution
Subscribes to the outputs: Feature vectors, Motivation vectors
Publishes the outputs: none
Status: Interface defined

Both components extend the IObserverOfPlans interface defined in the Slice definition for the Planning component. It allows the planner to deliver the generated plan to the components through the operation *setPlan*. Both components subscribe to the feature and motivation vectors. At any given time however, only one of them is in the process of generating a goal concept, sending it to the planner and awaiting a plan in order to pass it to Execution. The main difference between the two design components is that DesignOfExperiments provides a very specific service (namely that of the design of optimum experiments). The GoalDesign component however, has an additional managerial role as it is in charge of orchestrating the loop at all times when an experiment is not being performed and the learning process is not underway. In order to allow it to take the control of the loop back once DesignOfExperiments has completed its task an operation *takeControl* is found within its interface. Similarly, the interface for DesignOfExperiments contains the command *designExperiment*.

As they receive feature and motivation vectors, they must also implement the functions within the IObserverOfFeatureVector and the IObserverOfMotivationVector interfaces:

```
XPERSIF::ReturnCode setFeatureVector(XPERSIFDT::FeatureVector featureVector);
XPERSIF::ReturnCode setMotivationVector(XPERSIFDT::MotivationVector motivationVector);
```

Both these components are implemented as separate applications. The plans which are generated must be sent to EXECUTION through these components. This allows them to orchestrate the MOTIVATION, EXECUTION and MACHINELEARNING components thus retaining control of the workflow.

The GOALDESIGN component generates concepts within threads and calls a helper function *setGoalConcept* to set the concept it has generated and *notifyOnGoalConcept* which calls the PLANNING component with the new goal concept as argument. Within the *setPlan* operation (which allows the planner to send a plan to the DESIGN components), a check for the state of the loop would immediately execute the plan in the default state. While it receives the motivation vectors, it checks for a value for the surprise trigger. If this occurs, it brakes the EXECUTION component (whose *brakeComponent* operation stops all current commands). It would also brake the MOTIVATION component and set the new state of the loop to 'SurpriseValidation'.

In this state, two separate plans are needed – one for recreating the scene and another for repeating the actions which caused surprise. As it has been receiving feature vectors, the goal states needed to send the PLANNING component may be extracted. As soon as the state is set to 'SurpriseValidation' in the *setMotivationVector* function, a thread is spawned which generates these two goal concepts. The thread sets both goal concepts and calls the *notifyOnGoalConcept* helper function as before. This function would check the state of the loop (now set to 'SurpriseValidation') and would proceed to ask the PLANNING component to generate a plan for the first goal concept (recreate the scene). The *setPlan* operation would check the size of the plan list (in this state there should be two). If the list contains just one plan, a second call to the planner is made with the second goal concept (repeat the actions). When the PLANNING component sets the second plan, a call to EXECUTION is made to execute the first plan and a counter which keeps track of the number of times this plan has been sent for execution is changed accordingly. When a notification of the completion of the *executePlan* command is received from EXECUTION (through the *updateCalledByExecution* operation), the alternating plan is sent (as determined by the counters for each plan). If the execution failed for whatever reason, the state of the loop transitions to the default state once again. If surprise is validated consistently for the predefined number of times, then GOALDESIGN sets the loop state back to 'Default' and resets all counters and lists before calling the DESIGNOFEXPERIMENTS command *designExperiment* which signals it to take over the orchestration of the loop.

5.4.2 Planning and Execution components

The PLANNING and EXECUTION components are implemented as separate applications. The versatility to use any planner is possible as the EXECUTION component uses a

versatile plan representation language. This enables any planner to generate a plan, encode it in the representation and send it to either of the design components for it to be sent for execution. The EXECUTION component then executes the given plan and monitors its execution.

Component name: PLANNING
Component type: Basic
Application: Planning
Slice definition: `Planning.ice` (in section B.13)
Organized by: LOOPMODEL
Organizes: none
Observes: none
Subscribes to the outputs: none
Publishes the outputs: none
Status: Complete placeholder (Implemented for states zero and one of the movability experiment)

The interface for the IPlanning component includes the single command:

```
XPERSIF::CommandReturnCode generatePlan(XPERSIFDT::GoalConcept goalConcept);
```

The *generatePlan* command which, in addition to taking a goal concept as argument, also takes the proxy of the caller (one of the DESIGN components) so that it may return the plan to them once it is generated. As a command, it would return immediately, and allow the design component which sent this command to receive notification (which is especially important should the plan generation process fail). Once this command completes, it sets the plan using the *setPlan* operation.

Component name: EXECUTION
Component type: Basic
Application: Execution
Slice definition: `Execution.ice` (in section B.14)
Organized by: LOOPMODEL
Organizes: none
Observes: ROBOTMODEL, RELOCATION, MANIPULATION, PERCEPTION
Subscribes to the outputs: none
Publishes the outputs: none
Status: Complete placeholder (Implemented for states zero and one of the movability experiment)

The PLEXIL language – developed by NASA and Carnegie Mellon University – is the chosen plan representation language. It is designed to be portable, lightweight, predictable, and verifiable, and at the same time it is very expressive [31]. Its versatility lies in its use of XML. While the language specification is available, the Universal Executer itself which includes the interpreter is not available (although attempts to make it open-source are being made). The EXECUTION component will thus include an interpreter for it. The IXecution interface is also very simple as it contains the single command:

```
XPERSIF::CommandReturnCode executePlan(XPERSIFDT::PLEXILPlan plexilPlan);
```

The interface for the EXECUTION component is essentially a PLEXIL-based plan representation which allows the plan to then be executed on a number of embodiments. This design decision was validated through a thought experiment which attempted to weigh the benefits and the drawbacks of a plan generated for execution on a specific embodiment and a plan which is abstract enough to be executed by any embodiment. The results of this experiment are presented in section 6.4.1. The idea was to develop a series of batch-like, macro recipes which could be executed and the status of their execution monitored. If a problem was encountered, the EXECUTION component would handle the error by checking if any recipes could be applied. For example, if the goal of a plan was to move a box to a specified location and an error occurs due to the box being ungraspable for a particular manipulator, or a complete failure of the manipulation component, the EXECUTION component may attempt to push the box to the location. If it fails to do so, the error would be propagated upwards for possible replanning (indirectly through the DESIGN components). These 'recipes' may be found in Appendix D.

5.4.3 Feature selection component

The FEATURESELECTION component is responsible for deciding which features are relevant and should be extracted during the execution of a plan. This is necessary as the number of features which may be extracted may be quite large and their extraction may be time-consuming. The selected features are then extracted by the FEATUREEXTRACTION component. Once extracted, vectors are produced and these must be processed by other components. Other than its interfaces, this component is treated as a black box. Even the specification of an interface is a difficult task as it is not known what it may require in order to carry out its responsibilities.

Component name: FEATURESELECTION
Component type: Basic

Application: none
Slice definition: `FeatureSelection.ice` (in section B.15)
Organized by: LoopModel
Organizes: none
Observes: Relocation, Manipulation, Perception
Subscribes to the outputs: Motivation vector
Publishes the outputs: none
Status: Not implemented

Numerous possibilities have been considered. It may be that FeatureSelection receives the motivation vector in order to make use of surprising predicates (e.g. curiosity was high when the distance to an object was x; so it would ask FeatureExtraction to continue extracting the distance between the robot and the object). If the motivation vector is required, this component would need to implement the IObserverOfMotivationVector interface.

It may be that the MachineLearning component specifies the features it wishes to see in the feature vectors which it eventually uses to learn from. In this case, it could set the features which are to be extracted, by using the interface of the FeatureExtraction meta-component directly.

Yet another possibility is that the choice of features to be extracted is dictated by the DesignOfExperiments or GoalDesign components. The idea being that the designer of an experiment would design the experiment with the hopes of making specific measurements in the observation process and thus be in the best position to specify these measurements (features). Once again, this may be done by the design components directly.

Perhaps the simplest form this component may take is to choose features based on the action being executed. For example, if one is relocating, poses, angles and distances between the robot and other objects might be worthwhile to extract; in the case a grip command is being executed, then perhaps the sensor readings pertaining to manipulation are in order. In this case, FeatureSelection would need to subscribe to observe the various robotic hardware components.

5.4.4 Feature extraction components

With the goal of separating the execution and logging processes, two variants of the meta-component are used. Specifically, the RobotFeatureExtraction meta-component is used for the execution process while the FeatureExtraction meta-component provides feature vectors for various components within the loop. Both

are meta-components as they use the VISION component internally in order to provide their services. Both these variants are presented here starting with the FEATUREEXTRACTION component.

Component name: FEATUREEXTRACTION
Component type: Meta-component
Application: FEATUREEXTRACTION
Slice definition: `FeatureExtraction.ice` (in section B.16)
Organized by: LOOPMODEL
Organizes: Vision (two instances - one to process the robot image and the other to process the overhead camera image)
Observes: all components which it organizes
Subscribes to the outputs: Vision vectors
Publishes the outputs: Feature vector
Status: Complete (for the current set of features – which don't require vision)

Logging is handled by a single instance which is started by the LOOPMODEL component. It handles all available camera images and the raw sensor data received from the robotic hardware. The FEATURESELECTION component presented in section 5.4.3 is responsible for selecting the relevant features which should be extracted and delivered within the feature vector (by default, the FeatureExtraction component provides the time, objects and their poses). The IFeatureExtraction interface is summarized here:

```
XPERSIF::ReturnCode setFeaturesToExtract(XPERSIFDT::FeatureList featureList);
XPERSIF::ReturnCode addFeatureToExtract(XPERSIFDT::FeatureToExtract featureToExtract);
XPERSIF::ReturnCode removeFeatureToExtract(XPERSIFDT::FeatureToExtract featureToExtract);
```

The VISION component enables the extraction of features from a camera image. By default, objects and their poses (in robot-centric coordinates) are extracted. This is the case even when an image from an overhead camera is being processed. Additional features such as the area a detected object covers on screen may also be obtained if needed. A viewing client is needed as it acts as a buffer for the images. The VISION component is then able to ask for a frame once it has completed the processing of a previous one. In the case of a simulated experiment, the TeleSimView client is used (detailed in section 5.6.2).

Component name: VISION
Component type: Basic
Application: VISION
Slice definition: `Vision.ice` (in section B.18)
Organized by: FEATUREEXTRACTION and/or ROBOTFEATUREEXTRACTION

Organizes: none (but starts and exits the TeleSimView client)
Observes: none
Subscribes to the outputs: image from TeleSimView Client
Publishes the outputs: Vision vector
Status: Complete placeholder

The IVision interface is summarized here:

```
XPERSIF::ReturnCode addFeature(XPERSIFDT::VisionFeature visionfeature);
XPERSIF::ReturnCode removeFeature(XPERSIFDT::VisionFeature visionfeature);
```

The FeatureExtraction component uses the low-level features previously extracted by the Vision component along with raw sensor data obtained from the robot embodiment in order to extract high-level features (e.g. velocity, distance, etc). The feature vector is supplied as a data structure which allows the various subscribers to format the vector to their own specification.

The various data which it processes arrive at varying frequencies. This adds a level of complexity to the processing task. As it needs to receive the list of low-level features extracted by the Vision component, it implements the following function within its IObserverOfVisionFeatureVector interface:

```
XPERSIF::ReturnCode setVisionFeatureVector(XPERSIFDT::DetectedObjectList detectedobjectList);
```

The sequence diagram shown in figure 5.4 shows the initialization process of the FeatureExtraction component in the context of the whole loop in addition to the cyclic process of receiving data and producing feature vectors. While, the diagram displays these processes for the simulated case, the flow is virtually identical in the physical setting (the difference being the existence of the XPERSim simulator and the use of the TeleSimVew client in place of the TeleView client). RobotModel is seen starting the Perception component, having already started the Manipulation and Relocation components. The Perception component then starts the Camera subcomponent and configures it through a series of calls. Once this is done, it signals the Camera subcomponent to attach to the XPERSim server.

Meanwhile, the FeatureSelection component is started followed by FeatureExtraction. FeatureExtraction must first obtain a list of cameras from the Perception component (it may be the case that the robot is equipped with stereo-vision) in addition to obtaining the camera information used by the Observation component. The FeatureExtraction component starts two instances of Vision in

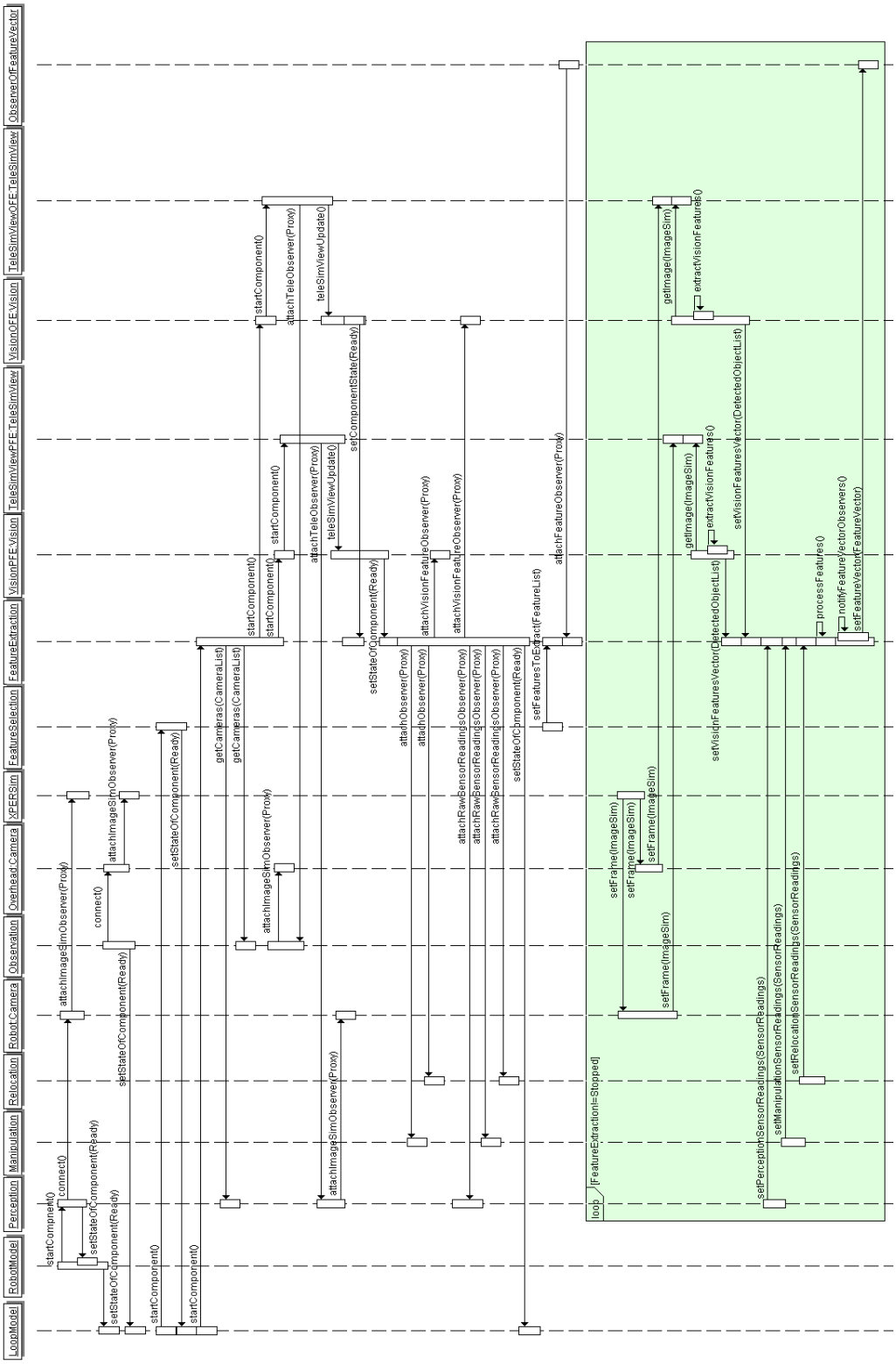

Figure 5.4: A UML sequence diagram showing the control flow relating to the FeatureExtraction components.

the case where both an overhead camera and the robot's camera are used. The VISION components in turn start TeleSimView clients (one for each camera). TeleSimView subscribes to receive the simulated robot view from PERCEPTION which in fact forwards the client's proxy to the CAMERA subcomponent (as it is the image provider). Once the VISION component is ready, the FEATUREEXTRACTION component subscribes to the VISION component's own detected objects' list. This is the case for the physical setting. Optimizations made within the simulated setting, necessitate that only a single VISION component is started with a single TeleSimView client.

The FEATUREEXTRACTION component now subscribes to each of the robotic hardware components' raw sensor readings and to observe them (i.e. to receive command status information). Having done this, the FEATUREEXTRACTION meta-component itself will set its status to: Ready. FEATURESELECTION may now set which features should be extracted. At this point, any component wishing to observe the feature vector may subscribe with the FEATUREEXTRACTION meta-component to receive it.

While the meta-component is not stopped or braked, it is extracting, processing and sending data. This is seen in figure 5.4 as a cyclic process of receiving images from the simulator through the TeleSimView client, their processing in order to extract low level vision features, the delivery of these low-level features to the FEATUREEXTRACTION component which subsequently processes them (and the raw sensor readings it receives) to extract high-level features and fill the data structure for transmission through the meta-component's interface.

With the implementation of this meta-component and the FEATURESELECTION component, the requirements depicted in the use case for making observations (figure 3.3) are met.

As mentioned previously, for each robotic embodiment within the loop, the ROBOTFEATUREEXTRACTION meta-component is used by its PERCEPTION component 5.3.4 (to service the execution process). If an overhead camera is also used, then an additional instance of this meta-component is used by the OBSERVATION component5.3.6 (also to service execution).

Component name: ROBOTFEATUREEXTRACTION
Component type: Meta-component
Application: RobotFeatureExtraction
Slice definition: `RobotFeatureExtraction.ice` (in section B.17)
Organized by: OBSERVATION and/or PERCEPTION
Organizes: Vision

Observes: Vision
Subscribes to the outputs: Vision vectors
Publishes the outputs: none
Status: Complete (for the current set of features – which don't require vision)

The initialization process of the RobotFeatureExtraction meta-component follows the process of obtaining the image from it (as is shown for the Perception component's camera). A major difference between RobotFeatureExtraction and FeatureExtraction is that no feature vectors are produced by RobotFeatureExtraction. They do, however, share the set of operations which provide the features. In fact, the interface of Perception and Observation include the set of operations allowing basic features such as the poses of objects or the distance between objects to be queried by other components. The use of RobotFeatureExtraction is detailed in section 5.3.4 and 5.3.6.

5.4.5 Prediction component

The Prediction component is responsible for creating models of the world. These models are used by the Surprise component which compares them with the observations (feature vectors) in order to detect discrepancies (behaviours that may not be explained by the prediction model). These models may be represented as first order logic clauses (as is the case currently) or in a much more complex representation such as prediction trees. The generation of these models is performed before the loop starts and stored in the Knowledgebase (currently, the models are generated manually and stored). In the future, the Prediction component is expected to use feature vectors in order to improve its models or even generate new ones. This is currently being researched. The architecture itself already provides for the online functioning of this component. For now, the predictions used by the Surprise component are obtained from the Knowledgebase. This component currently implements the IObserverOfFeatureVector interface in order to receive the feature vectors.

Component name: Prediction
Component type: Basic
Application: none
Slice definition: none
Organized by: LoopModel
Organizes: none
Observes: none
Subscribes to the outputs: Feature vectors
Publishes the outputs: none
Status: Not defined or implemented

5.4.6 Motivation components

The MOTIVATION meta-component generates the motivation vector which is used by others within the loop. It uses the services of the SURPRISE and CURIOSITY components to achieve this.

Component name: MOTIVATION
Component type: Meta-component
Application: Motivation
Slice definition: `Motivation.ice` (in section B.19)
Organized by: LOOPMODEL
Organizes: SURPRISE, CURIOSITY
Observes: all components it organizes
Subscribes to the outputs: Feature vectors, Surprise data structure, Curiosity vectors
Publishes the outputs: Motivation vectors
Status: Complete for use with the SURPRISE component

The diagram in figure 5.5 depicts the initialization process of the MOTIVATION components and the cyclic process of receiving feature vectors and producing motivation vectors.

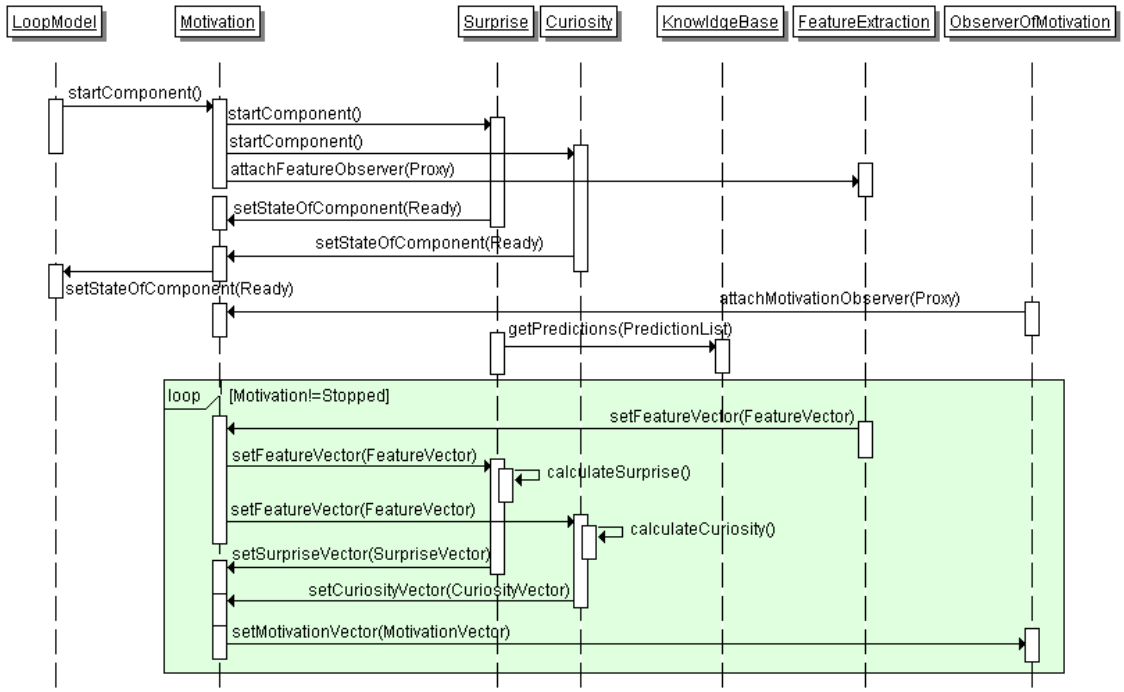

Figure 5.5: A UML sequence diagram showing the control flow relating to the Motivation components.

In addition to the operations and commands necessary for the fulfilment of its organizational responsibilities, the following functions of the IMotivationVectorSubject interface are included:

```
XPERSIF::ReturnCode attachMotivationObserver(IObserverOfMotivationVector *);
XPERSIF::ReturnCode detachMotivationObserver(IObserverOfMotivationVector *);
```

The SURPRISE component offers a computational model of surprise that enables the robotic learner to experience surprise and start the acquisition of knowledge to explain it [21]. In order to calculate surprise, the feature vectors which the MOTIVATION meta-component obtains are passed along to this component. Additionally, it requires the prediction models which specify how the world is expected to behave. These are obtained from the KNOWLEDGEBASE as the current logic of the loop dictates that they are produced prior to the start of the loop.

Component name: SURPRISE
Component type: Basic
Application: Surprise
Slice definition: `Surprise.ice` (in section B.20)
Organized by: MOTIVATION
Organizes: none
Observes: none
Subscribes to the outputs: Feature vectors
Publishes the outputs: Surprise data structure
Status: Complete placeholder

The CURIOSITY component is responsible for stimulating the robot to experiment. As part of the MOTIVATION meta-component, it receives the feature vectors and uses it to calculate a curiosity index for the discretized plane representing the environment within which the robot is situated. The resulting grid with the curiosity levels attached is then included in the motivation vector which is transmitted to subscribers.

Component name: CURIOSITY
Component type: Basic
Application: none implemented
Slice definition: `Curiosity.ice` (in section B.21)
Organized by: MOTIVATION
Organizes: none
Observes: none
Subscribes to the outputs: Feature vectors
Publishes the outputs: Curiosity vector
Status: Interface defined

5.4.7 Machine learning component

The MachineLearning component is responsible for interpreting the feature vectors and attempting to learn from them. Nothing is known about this component except that it receives the feature vectors and processes them as a batch in an off-line manner. The result of this processing is either a modified prediction model (so that a surprise is explained) or a new notion. The format of the feature vector itself remains an issue, as does the representation of the notion.

Component name: MachineLearning
Component type: Basic
Application: none implemented
Slice definition: `MachineLearning.ice` (in section B.22)
Organized by: LoopModel
Organizes: none
Observes: none
Subscribes to the outputs: Feature vectors
Publishes the outputs: none
Status: Interface defined

Currently, the feature vector is delivered to the various components as a structure. Naturally, this can be formatted in any manner by simply changing the data type definition. As it is distributed to a variety of components, it is important that the format be developed in close co-ordination with the component developers. As this component subscribes to receive feature vectors, it implements the IObserverOfFeatures interface. The IMachineLearning interface currently includes the following command:

```
XPERSIF::CommandReturnCode learn();
```

This is implemented as a command in order to allow DesignOfExperiments to query the execution status and receive notifications.

5.4.8 Knowledgebase component

The Knowledgebase provides a central store in which a priori knowledge is stored. This currently includes the predictions which have been generated offline and the models of objects within the world (e.g. redCube).

Component name: Knowledgebase
Component type: Basic
Application: none implemented
Slice definition: `Knowledgebase.ice` (in section B.23)

Organized by: LoopModel
Organizes: none
Observes: none
Subscribes to the outputs: Feature vectors
Publishes the outputs: none
Status: Interface defined

It is foreseen that the knowledge base will act as the central memory of the system. This would entail a change in the interface to allow the retrieval of items by more than just their property (e.g. by time). This component has the fuzziest state, as a representation of the knowledge has so far remained unspecified. In general, the interface implemented is based solely on the knowledge of what is stored within (predicates, models), in addition to the interfaces specified by the basic component-model.

```
XPERSIF::ReturnCode updateHypothesis(XPERSIFDT::Hypothesis hypothesis);
XPERSIF::ReturnCode addNotion(XPERSIFDT::Atom notion);
XPERSIF::ReturnCode getModelInfo(XPERSIFDT::ModelID modelID, out XPERSIFDT::Model modelInfo);
```

5.5 Software base applications

A base application has been implemented which facilitates the process of creating a new application. It contains all the current Slice files (which should always be updated from the XPERO SVN sever used for version control), the Server class and the ApplicationCom class. It also contains the `application.xml` file which is used by the Ice Grid service to register the application with the node. This section details these contents.

For the sake of flexibility, the repository of Slice files for the various components is included within the base application. Additionally, a separate Slice definition for a basic application has been specified. The file `ApplicationCom.ice` found in section B.3 specifies the IApplicationCom interface which includes the functionality to shut down an application.

The server class (currently implemented in C++ as with the rest of the project) contains the entry point to the application with the main function. The Ice Application is defined and the communicator is initialized within this class. The adapters to the components within the application are activated here. The application is then ready and waiting for shutdown.

The `application.xml` file provides the configuration of an application and its adapters running at an Ice Grid node. The file contains such information as the name of the node at which it will run, the path and name of the executable file, the type of start-up (e.g. on-demand) as well as a list of adapters to various objects and their properties. This is

the minimal skeleton which any application needs. Appendix C contains instructions on setting up an application on an Ice Grid node.

5.6 Experimentation via distributed simulation

This section details the efforts made to distribute the simulated images for tele-observation (the use case for which was presented in figure 3.6). Although the implementation is specific to the XPERSim simulator, the same approach could conceptually be used for other simulators.

Previous work to distribute the simulated images allowed an image to be requested and transmitted through a synchronous RPC call. The focus at the time was on providing the image as part of the feature vector and not tele-observation per se. A number of issues precluded the use of the same method for true real time tele-observation of the experiment:

- The presence of a bottleneck in obtaining the rendered image from the GPU (Graphics Processing Unit) to the CPU which makes the process of simply obtaining the image a time-consuming affair.
- The transmission of the image itself takes time.

It should be noted that these issues made infeasible the real time or quasi-real time tele-observation of the experiment by even one single client. In order to facilitate scalability, bottlenecks must be avoided.

The solution presented here, which bypasses this bottleneck, uses a proven methodology (often implemented in multi-player games) which involves moving the rendering of images from the server-side to the client-side by sending out a subset of the scene information to ensure that all clients are operating synchronously [14], thus drastically reducing the amount of data being transmitted. This is possible due to the scene-oriented nature of the XPERSim simulation. The Ogre 3D rendering engine (the simulator's graphics engine) uses scene-graphs to represent hierarchies, which simplifies the processing of objects or groups of objects. A scene-graph consists of nodes (with parent nodes and child nodes). If a parent node is translated or rotated, this transformation is applied to the child scene nodes as well. An example of such a hierarchy is shown in figure 5.6 where the Khepera robot's embodiment is depicted as a scene-graph.

The latency resulting from the distributed nature of the application is ameliorated by sending the node information from the simulator while the client is rendering the previous one – i.e. the server does not wait for the client to request the image but sends it continuously once it has subscribed. The method described above to distribute a

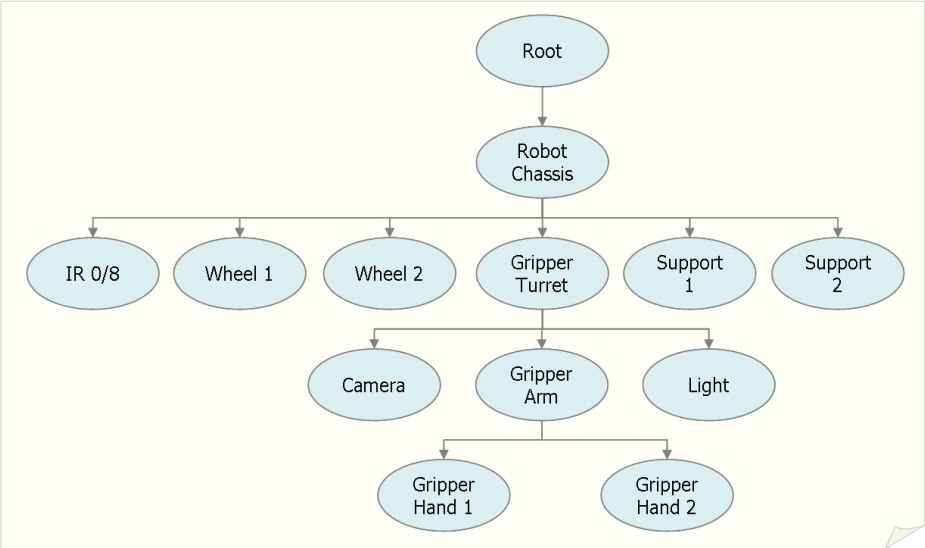

Figure 5.6: A scene-graph of the Khepera robot embodiment.

simulation to multiple clients is implemented here by decoupling the physics and graphics engines from XPERSim to create an XPERSim Server (just the physics engine) and a TeleSimView client (just the graphics engine). The XPERSim Server sends out the positions and orientations of all scene-nodes to the clients who simply transform the specified nodes to the specified positions and orientations and in so doing produce the same scene in an efficient and real time manner. In the refactored implementation of the XPERSim simulator, no distinction is being made between parent nodes and child nodes. It is recommended however that this distinction be made as it would reduce the number of nodes whose data needs to be transmitted (transmit parent nodes only and nodes which can be moved separately from the hierarchy – a gripper for example which, despite being a member of the robot node, may be moved on its own). The details of this implementation are described below.

5.6.1 XPERSim Server

As mentioned above, the XPERSim Server is now solely responsible for calculating the dynamics of the simulation and for their distribution to the clients. The separation of the two engines was straightforward due to the modular structure of the simulator. The first step was the removal of a windows-specific thread and mutex implementation and its replacement with Ice threads and mutexes. Previously, every rendered frame would step the simulation by 0.05 seconds (5 x 0.01 seconds). With this link to the rendering of a frame gone, the speed at which the simulation proceeded much faster. Various methods for transmitting the node information were evaluated (the results are presented in section 6.2).

The server contains two object adapters. One adapter is used to communicate with the robot (send commands to and receive data from it). The other adapter is used to send out the scene-nodes. During the start-up of the simulation and the creation of the ODE bodies, the information pertaining to the Ogre-scene is accumulated. This information is stored in a container structure which is requested by the tele-observer (the camera subcomponents). As the scene itself does not change, the same node information is sent out to both the observation and perception components. The same scene is rendered from each camera's position. As the robot's camera is attached to it, it will automatically be moved when the robot moves. If a pan/tilt camera is used, then its position could be sent out as a node.

In an effort to further reduce network latency, a one-way invocation is used to send the new frame. This can in fact be quite expensive when many such small messages need to be sent. This is because the run time taps into the OS kernel for each message and because each of these messages is sent out with its own message header [20]. To ameliorate this problem, batched one-way invocations are used. This allows the Ice run time to buffer these small messages until the XPERSim Server explicitly flushes them.

Originally, it was envisioned that the parametrization of XPERSim would be done through an XML file. This would allow the client to send the setup for a new experiment without necessitating the recompilation of XPERSim. The limited number of scenarios and the low frequency at which these scenarios are changed dispenses with the need for the XML parametrization and makes it equally efficient to choose precompiled setups.

5.6.2 TeleSimView Client

The TeleSimView client is used to view the simulated scene. With the same node information, the view from both cameras is rendered. The subscription to receive the node information is made with the hardware components: Perception or Observation. This process may be seen in figure 5.4.

The TeleSimView client only requires Ogre (and its dependencies). Ogre itself has always been cross-platform compatible. The source code for a project running under Windows could not previously be compiled and used directly on other platforms however due to the use of Windows-specific libraries handling events and key input. With the release of Ogre version 1.4.6 (a.k.a. 'Eihort'), this problem is now solved with the use of the Object-oriented Input System (OIS) platform.

The TeleSimView client implements the TeleSimView interface which allows the vision component to get an image from it. A two-way invocation to the Perception (or Observation) component fetches the scene which will be created and subsequently

updated. The creation of the scene involves the creation and attaching of nodes, their positioning and the creation of such basic scene items as the plane, lights, and sky. Once this has been done, the client would use the IXPERSimView interface which is extended by the PERCEPTION and OBSERVATION components in order to attach as an image-observer. As soon as this is done, the images will be transmitted to it from the relevant camera subcomponent (the image-provider).

The following chapter presents the results of a number of experiments which compared and evaluated the previous implementation and the refactored implementation presented here.

Chapter 6

RESULTS AND EVALUATION

This chapter presents the results and uses them to evaluate this work. In the first section the evaluation of the framework and architecture is presented. This is followed by the evaluation of the distributed simulation implementation. Lastly, the results of a number of thought experiments pertaining to the initialization process for the loop and the level of abstraction of a plan are presented.

6.1 The XPERSIF framework and architecture

The collection of UML use cases presented in Chapter 3 provide a valuable specification for the system's functional requirements. This collection of use cases facilitated the development process of the architecture. They were also useful for validating the system by providing tests with which to check that the system properly and completely implemented the use cases. To satisfy the use cases, the application logic provided by the partners must be integrated in an efficient manner. The framework's validation is accomplished by providing placeholder applications in order to mimic the flow of data and control within the cognitive loop. The results show that the design and the implementation of XPERSIF satisfy the use cases. A summary of the use cases and the validation of the system through them is presented here.

1. **Use case for feature selection, extraction and processing**: The design of the interfaces for the Vision, FeatureExtraction and RobotFeatureExtraction components enables this use case by providing the necessary functionality. The data types specified for these components and the use of the basic communication patterns presented in section 4.3 (e.g. to subscribe to raw sensor data or the image from the cameras) contribute to the efficiency of the system. The implementation of the application placeholders for the Vision, FeatureExtraction and RobotFeatureExtraction applications and their testing as seen in the experiments presented for the simulated setting in section 6.2, additionally prove the ease of use and the efficiency of this system.

2. **Use case for plan execution**: This use case was enabled through the careful refactoring of the robotic hardware components' interfaces. This provided the Execution component with high-level commands which can be executed on mobile manipulators and enabled a specific embodiment to be abstracted away.

This use case includes the use case which validated the observation process (presented above). The design of the framework in providing monitoring information through the notification mechanism (presented in Chapter 4) completes the validation process of the system.

3. **Use case for the cognitive loop**: The concrete experiment for movability (presented in Chapter 3) provided a use case for the loop. This use case guided the design of the architecture. The control flow diagrams (seen in figures 4.12, 4.13, 4.14 and 4.15) for the system correspond to the various use cases included in this large use case and serve to validate it by demonstrating that the functionality necessary for enabling the loop as a whole exists within the design.

4. **Use case for distributed simulation**: This use case has been validated by the implementation of a solution for distributed simulation. The results for this use case are presented in section 6.2.

An additional contribution was the development of a series of use cases for the partial plans (found in Appendix D). The validation of the design of the hardware components' interfaces using these use cases is presented in table 6.4.1.

Generally speaking, the evaluation of software integration frameworks is difficult to measure quantitatively [9]. Often the evaluation is done qualitatively against various criteria such as flexibility, reusability, scalability, ease of use, level of documentation, and development time. Nevertheless, conclusions can still be drawn from such an evaluation.

Flexibility of the framework was the core criterion in the development process. It is manifested not only by the ease of changing implementations independently of the abstract interfaces (e.g. using different planning algorithms beneath the same interface) but also by the ease with which components use each other's services. The flexibility was enhanced by the use of the organizational components which centralize the point at which changes might need to be made. Even a change of interfaces would be simple enough to propagate. The flexibility was further enhanced through the adherence to the SOA principle of service-stateless. The services offered through the component interfaces are at a level of granularity which enables their use under different control flow scenarios than the concrete example presented in Chapter 3.

Reusability has been achieved by allowing various existing implementations of an application to be reused beneath the interfaces. Numerous instances of components (such as VISION) also contributed to the reusability of a component.

Scalability has been ensured through the use of component-based approach and through the consistent use of simple and efficient communication patterns.

Ease of development was facilitated by the use of a single basic component model (augmented as need be for organizational components). The simplicity in design of this basic component model, which is nonetheless robust and efficient in propagating information through the various layers, is an achievement in itself. It allowed the system to be easily debugged and facilitated error handling which results in a *robust* system. The implemented base application also contributed to the ease of integration of components and applications. This ease of development also contributed to the *extensibility* of the architecture. The framework was designed with future needs in mind (e.g. the workbench, the use of stereo vision, of multiple robots, etc). This includes the facilitation of implementing a change in the experimentation process from the one presented in the concrete movability experiment in Chapter 3. The use of the GoalDesign and DesignOfExperiments modules to orchestrate the loop allows such changes to be confined to this component. The remaining components' interfaces are abstract enough to not need amendments.

Documentation is the Achilles heel of programming [26]. The work was accurately and consistently documented at a variety of levels including the source code, installation and user guides for the various versions which have so far been released. This holds for both the XPERSim and the XPERSIF projects. Additionally, a tutorial exists for the addition of robotic embodiments to XPERSim. The Slice definitions are all similarly documented in Doxygen style and are annotated with preconditions and postconditions. The Ice middleware too, has a high level of documentation which is both extensive and easy to reference.

The use of *Coding standards* have ensured that the source code itself is readable and consistent.

Sliding autonomy [8] is, in the case of this work, a valuable criterion, as the evaluation of the functional performance of the architecture must be carried out from the viewpoint of both the researcher and the robot (the two XPERSIF users), and often must be carried out from both points of view simultaneously. The ability to use XPERSIF under varying levels of autonomy is a necessity. The use of the LoopModel component to parametrize the experiment and the enabling of placeholders for the application (e.g. allowing a pre-generated plan to be used through the interface) provided this varying degree of autonomy.

The first integration of the simulator and robot control applications into the XPERSIF framework provided the initial test case for the framework. During a research camp for the XPERO project, the next integration of an ad hoc system (running under a Linux platform) with the existing robot control application (running under the Windows

platform) and the framework proceeded quickly and easily. It was then used to seamlessly connect and send commands to the RobotControl application. These commands were then executed in the simulated setting using XPERSim. Observations were made and sent back through the ad hoc system, and traces were generated for the machine learning process. The results validated both the ease of use criteria and the scalability criteria. Subsequent refactoring of the components encapsulating the robotic embodiment proceeded smoothly and with great ease, once again proving that the framework and architecture are flexible.

6.2 Distributed Simulation

A re-examination of the use case and requirements for distributing the simulation can be summarized with the following criteria:

1. Tele-observation (real time or quasi-real time) enabling all research teams to follow an experiment.
2. The use of the image in extracting features for execution of a plan.
3. The use of the image in extracting features for feature vector generation.

The results presented below show that these requirements have been met in an efficient manner. For the experiments performed for the evaluations in this section, the following experimental setup using four nodes were setup on four machines. The details of this setup are presented in table 6.1.

While all four nodes ran under the Microsoft Windows XP operating system, this does not preclude any of the applications from running on other platforms. This implementation was simply developed under the Windows platform. In fact, one motivating factor for the distribution of the simulation through the decoupling of the XPERSim simulator into its core components was to enable the same TeleSimView client to be compiled under any platform (as detailed in section 5.6.2). The use of the Ice middleware enables cross-compatibility of the remaining applications within the loop.

The first integration of the XPERSim simulator into the XPERSIF framework provided the image's color pixel values, in the BGR format, as a sequence of integers. In addition to the image itself, the width and height of the image, as well as the time to which it belongs, were also sent. A set of experimental evaluations was carried out to measure the time in seconds which is needed to receive a new image of size 416 x 600 pixels. The machines running nodes BRS and UVR were used in this initial experiment. For evaluating the speed of the tele-observation process in the local setting, the client

Node	Specification	Network Speed (Mbps)	Running Servers
BRS	Pentium 4 3.40GHz 2GB RAM ATI RADEON X800 GTO Windows XP SP2	18-24	RobotControl OverheadCameraControl Surprise XPERSimServer
TUW	Pentium 4 3.20GHz 2GB RAM ATI MOBILITY RADEON X600 Windows XP SP2	18-24	Vision RobotFeatureExtraction TeleSimView
UVR	AMD Turion 64 Mobile 1.79GHz 1GB RAM ATI Radeon Express 200M Windows XP SP2	18-24	Client starting loop LoopModel Execution TeleSimView
AUP	Pentium 3 0.8GHz 500MB RAM Intel Extreme Windows XP SP2	100	TeleSimView

Table 6.1: The specifications of the experimental setup used to evaluate the refactored implementation for distributed simulation.

initiating the RPC call for the image and the servers (XPERSim and the robot and overhead camera control application, as well as the registry) were all co-located at a single node (BRS). For the distributed setup, the image was received by the client running on the UVR node while the registry and servers ran on the BRS node.

A comparison of the performance of the previous XPERSIF-XPERSim implementation and the current one (client-side rendering) is presented in table 6.2. An analysis of these measurements allows an evaluation of the implementations in terms of the number of frames that may be received in one second (fps) to be made. Table 6.2 shows the results of this evaluation.

Setup	Previous implementation		Refactored implementation	
	seconds	fps	seconds	fps
Local	0.1846	5	0.0062	160
Distributed	9.5346	< 1	0.0046	217

Table 6.2: The time in seconds needed to receive an image by a single client in both the local and distributed settings is shown here alongside the number of fps which could be delivered given these measurements.

In the case of the previous implementation of XPERSim, the values include the roundtrip time of the call, which included the transmission of the pixel values of the

image. In the current implementation, the use of the observer pattern removes the need for the client to make such a request for each and every frame, opting rather for a single subscription to the tele-observation service and the asynchronous delivery of the image (in the form of nodes, not pixel values).

While table 6.2 shows encouraging results for the refactored implementation for distributed simulation, it does so for a single client in both the local and distributed setup. The scalability of this implementation was evaluated by measuring the impact on the quality of the simulation by varying the number of subscribers to the tele-observation service. This detailed scientific evaluation validated the use of a batched one-way invocation for distributing the image.

Table 6.3 shows the measurements made when one, three, five and then ten clients are subscribed to the service. All experiments were repeated three times, measuring the time it took for 60 frames to be delivered to the TeleSimView client used by the VISION component for extracting detected objects (which are finally used by the OBSERVATION component in executing a plan).

An additional evaluation relates to the invocation method used to update a frame. As mentioned in section 5.6.1, a batched one-way invocation means that several one-way messages are buffered by the Ice core and then sent together in a single request on demand thus bypassing the overhead used up when sending many small messages independently. It was believed that a batched one-way connection between the image provider (CAMERA subcomponent) and the client would prove to be much more efficient. This expectation was validated by the results shown in 6.3. As mentioned previously, the CAMERA subcomponent delivers images to subscribers by spawning a single thread on connecting to the XPERSim server. This thread polls for a new frame. It then retrieves the frame and the list of tele-observers then notifies them in a loop. Through a number of additional experiments, it was determined that the optimum place to flush the buffer in order to dispatch the images was from within the function where the CAMERA subcomponent receives a new frame from the XPERSim server. This ensures that the number of tele-observers which must be notified does not greatly impact the delivery of the images. A smoother viewing of the simulation is thus obtained. This thread polls for a new frame. It then retrieves the frame and the list of tele-observers then notifies them in a loop. Through a number of additional experiments, it was determined that the optimum place to flush the buffer in order to dispatch the images was from within the function where the CAMERA subcomponent receives a new frame from the XPERSim server. This ensures that the number of tele-observers which must be notified does not greatly impact the delivery of the images. A smoother viewing of the simulation is thus obtained.

It is imperative that the delivery of the simulation to the clients does not disrupt the plan execution process. As the motion controller (used by the RELOCATION component) uses the overhead camera for localization, the timely delivery of images/scene-graph nodes which ultimately allow it to perform must not be jeopardized. In the case where plan execution failed due to the inconsistent delivery of the images, the symbol x is placed alongside the measurement.

Invocation method	Trial	1 client	3 clients	5 clients	10 clients
Two-way	1	0.0044 s x	0.0730 s	0.0318 s	0.1659 s x
	2	0.0065 s	0.0135 s	0.0221 s	0.1789 s x
	3	0.0055 s	0.0120 s	0.0302 s	0.1023 s x
	Mean	0.0055 s	0.0328 s	0.0280 s	0.1490 s
Batched One-way	1	0.0039 s	0.0039 s	0.0219 s	0.0227 s
	2	0.0023 s	0.0172 s	0.0128 s	0.0352 s
	3	0.0075 s	0.0036 s	0.0120 s	0.0448 s
	Mean	0.0046 s	0.0082 s	0.0156 s	0.0342 s

Table 6.3: The time in seconds between receiving two subsequent images using different invocation methods.

While plan execution proceeded smoothly using batched one-way invocations, the plan took much longer to execute (in terms of real-world time (not simulated time)). The flushing of the buffer forced all TeleSimView clients to render more frames than under the two-way invocation scenario where due to the longer time necessary to send the images to clients, more frames are missed (they arrive at the CAMERA subcomponent and are overwritten several times before the thread can retrieve them). This loss of frames causes the choppy viewing experience (where the robot appears to stop and then suddenly jump forward) when using the two-way invocations. As the images contain vital information for localizing the robot, the motion controller is unable to obtain up-to-date poses and the plan ultimately fails especially as the number of clients increases.

The Ice implementation of both two-way and one-way invocations use TCP/IP. If a two-way invocation fails, the thread invoking the *setFrame* operation in the TeleSimView client's interface is guaranteed an exception. This is not the case for one-way invocations. Moreover, if a batch one-way message is lost, all invocations within it are also lost. Despite this, batched one-way invocations are the most viable solution enabling scalability of the tele-observation service. A more serious issue which applies to all invocations using Ice may be seen when network connectivity is a problem. It is possible, for example, for the CAMERA subcomponent to block if for some reason a subscribed TelSimView client has network connectivity problems. This is because the Ice runtime attempts to establish the connection transparently and waits for an acknowledgment

that the connection has indeed been established. Unless a timeout is set, this wait will block the CAMERA subcomponent until it is resolved. This issue is not specific to the use of one-way invocations but exists for all two-way invocations as well. This problem will no longer be an issue starting with the release of Ice version 3.3. It is important to note that the shutting down of a TeleSimView client at any time during the tele-observation process does not disrupt the process for any other clients as the exit sequence includes the call to unsubscribe from the CAMERA subcomponent, thus preventing the CAMERA subcomponent from blocking.

It is worth noting that the size of the image to be rendered is now inconsequential. As nodes are being sent and not an image, it is the number of nodes within a scene that impacts the time and not the image size. For the test case above, 15 nodes were transmitted (representing the Khepera robot and the four cubes used for the movability experiment). Using this information, the scene may be rendered from the viewpoint of any number of cameras.

The second and third requirements presented above have also been met efficiently. As all instances of the VISION application (those used for execution and those used for producing feature vectors) make use of the TeleSimView client, they benefit from fast access to images (as demonstrated through the results above). Moreover, the placeholder component (and application) which is implemented allows VISION to obtain the simulated image either as a file in 'png' format or as a series of nodes (just as they were sent from the CAMERA subcomponent). The current implementation of the placeholder for the VISION component is able to transform the positions and orientations of the nodes into robot-centric coordinates (except for the robot embodiment which is always represented in global coordinates) in order to provide them to the FEATUREEXTRACTION component (as it does under the physical setting).

6.3 The contribution of simulation to research

In this section we present results which support our argument that simulation has indeed sped up the pace of research within the XPERO project. It has aided the human researcher in exploring various scenarios simultaneously and provided valuable traces for use in streamlining the machine learning tools. Figure 6.1 provides a breakdown of the data generation process over the course of the XPERO project according to the methods used. The comparison shows that around 90% of the traces have been generated using a simulator.

Figure 6.1: An overview of the number of traces generated throughout the XPERO project (04/2006 to 02/2007) and the methods with which they have been generated.

6.4 Thought experiments

This section presents the results of a number of thought experiments (as in H. C. Orsted's *Gedankenexperiment*) performed at various stages of this work. Although the techniques which are used in these experiments are non-empirical in nature, their use is nonetheless of great benefit.

6.4.1 On plans for abstract vs. specific embodiments

The ability to execute a single plan on various embodiments is very useful, as the generation process would not have to be repeated each time the embodiment changed. While this relates to a plan's flexibility, it is not in the traditional sense used by Firby in the development of Reactive Action Packages (RAP) [13] where flexibility relates to the choice of various task plans for execution in order to reach a goal without the need to replan (although a similar hierarchical set of tasks / recipes is also used in PLEXIL). We have carried out thought experiments in order to ascertain whether the level of abstraction of a plan impacts the overall efficiency (and successful) execution of a plan with results which were highly useful in the design process of the interfaces to the components.

The results showed that making the assumption that all embodiments are mobile manipulators allows the generation of a plan using only information about the surroundings (as obtained from the knowledge base and/or current observations) and not the specific embodiment (e.g. push(redBox); approach(wall1)). In fact, criteria for intelligent systems proposed by [25] suggest that a robot's level of intelligence may be evaluated through the level of understanding of higher level, more abstract commands, and its ability to supplement the given command with additional information that helps

to generate more specific plans internally. Such abstract plans are passed to the EXECUTION component where specific values (such as length of a box, or the positions from which it may be gripped) are used to make the plan more concrete. A series of use cases for various plans were drawn up and an attempt was made to validate the possibility of abstracting away the particular embodiments.

With the results of this thought experiment, and the use cases specifying these batch-like, macro 'recipes', the interfaces for the robot embodiment components were revised in favour of more abstract ones. The skills class which forms the basis of the hardware abstraction layer (presented in section 5.3.1) was also updated to this end.

Naturally, not all embodiments (or groups of embodiments – e.g. mobile manipulators) are created equal in terms of their abilities. Manipulators, for example, offer varying degrees of freedom. An object which may be graspable using one manipulator may not be graspable using another due its configuration. Within the XPERO project, the objects are limited to boxes and spheres. This laboratory environment goes a long way in further simplifying matters. It ensures that even the simplest manipulators may grip the various objects. Table 6.4.1 presents the raw results of the thought experiment (the recipes) and how the embodiment and overhead camera interfaces satisfy their needs.

These results also helped in the specification of necessary information to enable the nodes presented above. This information relates to, for example, poses from which an object may be gripped, or the constraints of these poses (e.g. object is graspable only at a specific height). This information can be seen in figure 6.2.

The results point out the viability and effectiveness of generating abstract plans which become more specific (first by filling in the concrete metrics of an object at the EXECUTION component level and then by filling in the concrete metrics of a robot within the status and skills class of RobotControl).

6.4.2 On initializing the loop

This section identifies a number of parameters which should be made on initializing the loop in order to provide more flexibility in choosing various configurations for an experiment. In a way, the information necessary to configure an experiment is related to the information which is stored in the knowledge base.

As the XPERSim simulator has been integrated into the framework, the human researcher should specify whether the experiment is to be performed in a simulated setting or the physical setting. This may be done by setting the Xperiment mode . Additionally, the researcher should choose what hardware is used. For example, he/she

Node name	Implementation
drive to pose by rotating, moving forward	Calculate angle to rotate Relocation→rotateByAngle(angle) Calculate distance to move Relocation→moveForward() Loop until distance is reached Observation→getRobotPose(robotPose) Calculate elapsed distance Relocation→stopDriving()
roam	Loop Until surprised or Timeout: Generate random pose Relocation→moveAbs(pose)
approach approach to view (redBox) to contact (redBox) to grip (redBox) to push (redBox)	Calculate the angle to approach with a heuristic KnowledgeBase→getModelInfo(redBox) Observation→getObjectPose(redBox) Do calculation based on type, pose and heuristic Calculate Path to follow with the calculated angle) Calculate final pose (with object and robot info) Calculate way points (with robot info) Relocation→FollowPathAbs(calculatedPath)
push push from angle	Approach to push (RedBox) Relocation→MoveForward() Loop until stop condition is met Relocation→stopDriving()
grip object (redBox)	ApproachToGrip(redBox) Calculate grip pose KnowledgeBase→getModelInfo(redBox) Manipulation→moveTooltipPose(pose6D) Manipulation→grip()
carry object (redBox,Pose)	GripObject(redBox) Manipulation→prepareToRelocate() Relocation→moveAbs(Pose) Release Object (redBox) *or* Stack Object (redBox, blueBox) Calculate pose to release Manipulation→moveTooltipPose(pose6D) Manipulation→ungrip()
circumnavigate object (redBox)	Calculate circumnavigation way points (redBox) KnowledgeBase→getModelInfo(redBox) Observation→getObjectPose(redBox) Calculate circumnavigation path (waypoints) Observation→getRobotPose(robotPose) Choose closest way point Rearrange path

Node name	Implementation
move with object in view	Repeat Until surprised or Timeout: ApproachToView(redBall) Calculate angle to rotate using camera's field of view Relocation→rotateByAngle(angle) While Perception→isObjectInView(redBall) Relocation→moveBackwards() Relocation→stopDriving() Calculate angle to face object Relocation→rotateByAngle(angle)
follow wall	Repeat Until surprised or Timeout: Calculate wall way points KnowledgeBase→getModelInfo(wall) Repeat until all corners visited or Timeout Relocation→MoveAbs(WayPoint(i)) Calculate angle to face next way point WayPoint(i++) Relocation→rotateByAngle(angle)

Table 6.4: A breakdown of the number of traces used to streamline the learning tools and methods since the beginning of the project.

should decide if an overhead camera is used, which camera specifically, its position and specifications, etc. A similar decision should be made regarding the specific robotic embodiment which acts as the protagonist. The environment within which the experiment will be performed should also be parameterized. This might include the selection of objects (and their parametrization). It may also include other robots, in which case, they are treated as objects.

Other issues which need configuring include the selection of which components within the MOTIVATION meta-component are to be used (e.g. CURIOSITY, SURPRISE, NOVELTY, etc.) and their parametrization (thresholds for trigger notifications, etc). In the current setup for the movability experiment, values for the number of times to execute surprise-validation plans might be given. Similarly, the number of times to execute an experiment could also be parametrized. As could values for timeouts for executing a plan. A mechanism to ensure the necessary a priori knowledge (including the prediction models) are in the knowledge base prior to starting an experiment should be used.

The diagram shown in figure 6.2 presents an analysis of the physical setup of the loop in more detail.

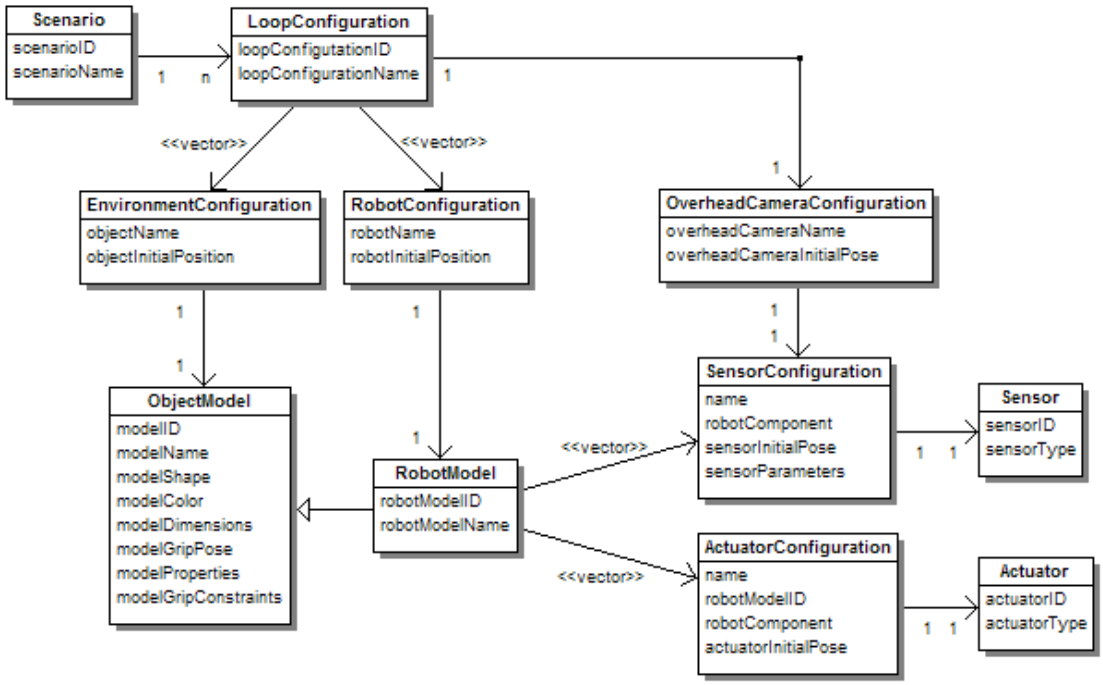

Figure 6.2: An analysis of the information needed to specify an environmental setup.

Chapter 7

CONCLUSIONS

7.1 Contribution

In this thesis we motivated and presented the design and implementation of a software integration framework for robot learning by experimentation. The framework itself (the component model and core data types) needs no prior knowledge of specific components and is thus flexible enough to be used by all components. The principles of the service-oriented approach have lent the resulting architecture the advantages associated with it in terms of composability, reuse, autonomy, loose-coupling and abstraction of services. The organizational components facilitate tracing, profiling and debugging of the system while meta-components allow components to be grouped together in order to provide an aggregate service (thus supporting composability). The system's design meets the needs of both the human researcher within the loop and the autonomous robot itself.

The resulting framework and architecture facilitate the integration process for developers through a flexible design which allows the various research groups to make decisions across the board from what data structures to use to the language and the platform. The framework minimizes user effort in integrating their components as well as in making changes once the implementation is complete.

It enables plug-and-play functionality of various implementations of software applications by providing abstract interfaces focusing on the service being provided and not the implementation. This abstraction of the applications also contributes to the applications' concurrent development. We have provided logging of various vectors as services available to any component and successfully implemented it in an efficient manner by separating this task from that of execution monitoring.

A collection of interface specifications has been compiled which should contribute significantly to the integration effort. This includes the specification of data types. While some of these remain in a preliminary state due to a lack in more detailed specification, others are in a much more evolved and stable state.

In addition to the available interface specifications, a number of components have in fact been fully implemented. Specifically, the hardware components (for the robotic embodiment and the overhead camera) have been revised and their implementation

completed. Finally, application and component placeholders have also been implemented and server to provide examples for the implementation of the three types of XPERSIF components.

We have addressed the problem of tele-observation by decoupling the physics and rendering components within the XPERSim simulator in a manner that optimizes computational power and harnesses the power of node-oriented scene-graphs, and thus reduced network latency. This has been further optimized by using one-way invocations for the dispatching of the node information. The decoupling of the XPERSim simulator's core components has significantly enhanced its usability by allowing the simulation to be viewed on multiple platforms and in an efficient manner. Results support our argument that there is certainly a great advantage to be had when simulation is used at the various stages of research

The framework in combination with XPERSim provides a means to validate plans by sending them to the simulator before using the physical robot. The framework will enable the acceleration of the experimental loop to streamline the learning tools as fast as possible. With the use of Ice Grid as a location service, it becomes possible to distribute the evaluation of an experiment to all designers working on the experimental loop quickly and easily.

7.2 Open issues

It will be necessary to make use of more Ice services, such as *Glacier*, to allow clients and servers to securely communicate through firewalls. The instantiation of dedicated instances of specific packages could be implemented using Ice sessions. Alternatively, object factories which produce these components might be created.

With regard to the FeatureExtraction component and other similar components that receive data at varying frequencies, a mechanism for synchronisation should be attempted. These components might wait to receive all data and fuse these before dispatching them, or they might fuse the data as they get it with the latest data they have. Alternatively, the framework itself might implement a global clock for synchronization purposes. This should then take into account the discrepancies between real and simulated time. The resulting system is then no longer an asynchronous one which may result in a loss of efficiency at some level.

The use of complex manipulators would require a more complex controller to be used. This controller would most likely need to work in coordination with the relocation drive. Real time issues will most likely emerge. This is a general robotics problem and out of the scope of the framework.

XPERSim-related open issues include the need for upgrading of the physics engine ODE to the latest stable version. Additionally, an upgrade to the latest Ogre distribution would also greatly enhance performance.

A continuing refinement of the service interfaces is necessary. The integration of the components and applications should commence swiftly. The KNOWLEDGEBASE is a key component on which there has so far been little progress. The lack of representation has been the biggest hurdle in the path of this component's specification.

7.3 Lessons learned

One of the most impressive lessons learned was that the process of requirements specification needs to be effected through frequent bilateral and multilateral meetings with the partners involved. The use of the UML standard for modelling the processes involved and the requirements themselves is extremely helpful and should be used in the future. The initial process involving the identification of the actors is one step which proceeded smoothly. The definition of interfaces was complicated as the application developers themselves were often unsure of the services they would provide and more specifically the data types they would use. Sometimes, even the frequency at which these services would be provided was simply unknown. More significantly, the logic and the flow of control within the loop is still to a large extent unstable.

This uncertainty has necessitated that the bulk of the effort with regard to framework (and architecture) design be aimed at maximizing flexibility. This has been achieved. While, in itself, this is a worthwhile characteristic to have, its implementation had notable impacts on the architecture (increasing the overhead for example and thus the weight of the system).

Due to these unstable settings, every time an interface definition is changed, all instances of the Slice file will need to be updated. The Ice Patch service partly facilitates this process and allows consistency to be maintained.

The use of off-the-shelf middleware has been very effective in facilitating the development process. The use of Ice as the chosen middleware has been validated through the implementations presented here. Moreover, it provides the language mappings necessary for the specific needs of the research groups in the implementation presented here. It has proven to be both efficient and robust, with an impressive learning curve and very good documentation and support through the forums.

The use of the waterfall method for software engineering, although criticized as being inflexible as the process only flows in one direction, was nonetheless necessary in the case

of this work. As this thesis is bound by a limit in time, it was in fact the only possible approach. With regard to the whole project, however, this model should be discarded in favour of more flexible agile techniques [24].

With regard to the use of XML for the initialization of the loop and the simulation, it was initially thought that XML parametrization would naturally provide the most flexibility. However, given the specification of the scenarios and the showcase and the limited number of setups, it should be possible to simply specify which scenario and embodiment (and the presence of additional hardware such as the overhead camera) at the beginning of the experiment. While it may be useful as a format for the knowledge base, its use could be limited to those components which truly benefit from it.

We believe that this type of project would benefit from a software testing framework. Due to the modular nature of the framework and the components which are used within, the testing would contribute greatly to the robustness of the overall architecture. This is especially important once the further integration of applications gets underway.

Finally, with respect to the reusability of the framework and the effort to maintain a thin framework towards this end, we believe this has been accomplished by the minimal use of the Ice middleware and the use of built-in communication mechanisms instead (for example, Ice's service for publisher/subscribe is not used). This is in itself a good thing. However, it may be that a tighter integration with the middleware might prove to be more efficient in terms of runtime.

7.4 Road map: From here to a workbench for robotic learning by experimentation

A potential goal of the project is to ultimately provide a workbench for robot learning by experimentation. This should allow the whole process to be specified, parametrized and started at run-time through a GUI. Additionally, the GUI should provide a variety of monitoring information.

The first task is the specification of the requirements which the GUI should fulfil. A list of features thus provides a starting point:

- Parametrizing hardware setup of the loop (even in the case of a physical experiment, it is necessary for the LoopModel component to know how many robots (and which models) are used; also, whether or not an overhead camera is used).
- Parametrizing the loop with 'optional' components. It is clear that while some components are an absolute necessity, others are not. For example, one form of motivation stimulus should be enough to trigger events. This could involve the choice of the application implementation (e.g. which planner).

- Choosing the level of autonomy for each component. This includes the user providing a service of the component if this is desired (e.g. a pre-generated plan).
- Choosing conditions to pause the flow within the loop (e.g. when a state transition occurs).
- Choosing a logging method (to a file/console).
- Monitor the loop (parametrizable: which values does the user wish to monitor).

The next step would involve the validation of a functional GUI, such as that provided by the "Orange" tool's Python-based libraries (currently the first candidate for the GUI). The support of Ice for Python should facilitate the integration process. The back end to the GUI is the LoopModel component which conveniently provides access to the entire flow of data, the components and their states. While the initialization and monitoring of the loop should be straightforward, methods that allow changes at runtime may prove to be more complex.

By providing this framework, we have brought the goal of a workbench of tools for robot learning by experimentation that much closer to reality and provided both autonomous robots and roboticists with a valuable and useful set of tools with which to address their problems.

BIBLIOGRAPHY

[1] M. Aicardi, G. Casalinoand A. Bicchi, and A. Balestrino. Closed loop steering of unicycle like vehicles via lyapunov techniques. *IEEE Robotics & Automation Magazine*, 2(1):27–35, Mar 1995.

[2] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research*, 17, April 1998.

[3] I. Awaad and B. Leon. Simulation of the robotic experimenter. Technical report, University of Applied Sciences Bonn-Rhein-Sieg, March 2006.

[4] I. Awaad and B. Leon. Xpersif:a component-based software integration framework for robot learning by experimentation. Technical report, University of Applied Sciences Bonn-Rhein-Sieg, July 2007.

[5] I. Bratko, J. Demsar, M. Guid, M. Mestnik, D. Suc, and J. Zabkar. Xpero: Learning by experimentation - deliverable 4.1: Gaining insights about properties of single objects, May 2007.

[6] Ivan Bratko. Initial experiments in robot discovery in xpero. In *ICRA 2007 Workshop on "Concept Learning for Embodied Agents"*, April 2007.

[7] R. W. Brockett. Asymptotic stability and feedback stabilization. In R. S. Millman R. W. Brockett and H. J. Sussmann, editors, *Differential Geometric Control Theory*, pages 181–191. Birkhauser, Boston, 1983.

[8] Jonathan D. Brookshire. Enhancing multi-robot coordinated teams with sliding autonomy. Master's thesis, Carnegie Mellon University, 2004.

[9] Paul Caspi and Oded Maler. From control loops to real-time programs. In William S. (Eds.) ristu Varsakelis, Dimitrios; Levine, editor, *Handbook of Networked and Embedded Control Systems*, Control Engineering, chapter From Control Loops to Real-Time Programs. A Birkhuser book, 2005.

[10] MSDNAA Editor. .NET framework essentials. Technical report, -, August 03 2001.

[11] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[12] V. V. Fedorov. *Theory of Optimal Experiments*. Probability and Statistics. Academic Press, Ney York, New York, 1972.

[13] R. James Firby. *Adaptive Execution in Complex Dynamic Domains*. PhD thesis, Yale University, January 1989.

[14] Ogre Forums. Streaming a scene rendered by a camera. Online at http://www.ogre3d.org/phpBB2/viewtopic.php?p=224646, May 2007.

[15] Brian Gerkey and contributors. *The Player Robot Device Interface*, 2005.

[16] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, May 2004.

[17] R. Grimes and Dr R. Grimes. *Professional Dcom Programming*. Wrox Press Ltd., Birmingham, UK, UK, 1997.

[18] O. M. Group. *Common Object Request Broker Architecture (CORBA/IIOP) Version 3.03*. The Object Management Group, March 2004.

[19] The Miro Group. *Miro Manual Version 0.9.4*, January 2006.

[20] M. Henning and M. Spruiell. *Distributed Programming with Ice*. ZeroC Inc., revision 3.2 edition, March 2007.

[21] Alex Juarez. A computational model of robotic surprise. Master's thesis, University of Applied Sciences Bonn-Rhein-Sieg, 2008.

[22] Alexei Makarenko. *The ORCA manual 2.7.0*, October 2007.

[23] Alexei Makarenko, Alex Brooks, and Tobias Kaupp. On the benefits of making robotic software frameworks thin. International Conference on Intelligent Robots and Systems Workshop, October 2007.

[24] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.

[25] E. Messina, J. Evans, and J. Albus. Evaluating knowledge and representation for intelligent control. In *2001 Performance Metrics for Intelligent Systems (PerMIS) Workshop*. IEEE CCA and ISIC, 2001.

[26] Anders Orebäck. *A Component Framework for Autonomous Mobile Robots*. PhD thesis, KTH, 2004.

[27] Erwin Prassler. Personal Communication, 2007.

[28] Russell Smith. *Open Dynamics Engine*, 2006.

[29] The OGRE Team. *OGRE Manual v1.2.0 ('Dagon')*, 2006.

[30] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar. Miro - middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation*, 18(4):493– 497, August 2002.

[31] V. Verma, T. Estlin, A. Jnsson, C. Pasareanu, R. Simmons, and K. Tso. Plan execution interchange language (plexil) for executable plans and command sequences. In *International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS)*,, 2005.

[32] P. Welkenbach, M. Heinisch, D. Liebhart, M.l Knings, G.Schmutz, M. Klliker, and P. Pakull. *Architecture Blueprints*. Carl Hanser, 2006.

[33] Wikipedia.org. Middleware. Online at http://en.wikipedia.org/wiki/Middleware (Accessed: May 17, 2007).

[34] V. A. Ziparo, A. Kleiner, B. Nebel, and D. Nardi. RFID-based exploration for large robot teams. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, 2007. to appear.

Appendix A

Typographical conventions

The following typographical conventions are used within this thesis:

`Filename.filetype`

ComponentName

InterfaceName

DataTypes

ClassName

FunctionName

Appendix B

Ice files

This appendix contains a listing of all the ice files containing the slice definitions used in the current implementation of XPERSIF.

B.1 XPERSIF.ice

```
// File XPERSIF.ice
#ifndef _XPERSIF_ICE
#define _XPERSIF_ICE

module XPERSIF{

    /**
    * Loop State
    */
    enum LoopState {
        Exploration,
        SurpriseValidation,
        Experimentation,
        Learning
    };

    /**
    * Component State
    */
    enum ComponentState {
        Unknown,
        Registered,
         Initialized ,
        Ready,
        Failed,
        Operating,
        Stopped
    };

    /**
    * Component ID
    */
    enum ComponentID {
        LoopModel,
        RobotModel,
        Manipulation,
        Relocation,
        Perception,
        Observation,
```

```
    DesignOfExperiments,
    GoalDesign,
    Planning,
    Execution,
    Vision,
    FeatureSelection,
    FeatureExtraction,
    Prediction,
    Surprise,
    Motivation,
    MachineLearning,
    Knowledgebase
};

/**
* The state of process completion of a command. Set by the command implementation.
*/
enum CommandTaskState {
    Running,          // The task is running
    Finished,         // The task is finished
    CommandAbort,     // The task could not be completed;
                                        // the component is in a permissable state
    CommandFatalError // An internal error occurred and the component
                                        // is in an unknown state
};

/**
* Information for the execution of an operation
*/
enum ReturnState {
    Success,                    // Successfully completed
    Abort,                      // The operation could not be completed but the
                                        // component is in a permissable state
    ContractViolation,  // Incorrect use of the operation;
                                        // the component is in an unknown state
    FatalError          // An internal error has occurred and the component
                                        // is in an unknown state
};

/**
 * System-wide identification of a command process.
 * This contains the ID of the component and is
 * created by the component on receipt of the command.
 *
 */
struct CommandTaskID {
    /** The system-wide identification of the component */
    short componentID;
    /** The number of the command (unique within the component) */
    int number;
    /** The type of the command */
    int commandType;
};

/**
```

```
 * Contents of a notification or "getCommandStatus()" reply.
 */
struct CommandTaskInfo {
    /** The system-wide identification of a task */
    XPERSIF::CommandTaskID commandTaskID;

    /** The state of the command. */
    CommandTaskState state;

    /** Description of the state of the task completion.
     * optional: The string can remain empty, but must be included.
     */
     string description;

    /** The forseen length to completion of the task */
    long durationToFinish;
};

/**
 * List of CommandTaskID.
 */
sequence<CommandTaskID> CommandTaskIDlist;

/**
 * List of CommandTaskInfo.
 */
sequence<CommandTaskInfo> CommandTaskInfoList;

/**
 * Return type for an operation.
 */
struct ReturnCode {
    /** State after completion of the operation */
    ReturnState state;

    /** Description of the state of the task. Can contain information on errors.
     * optional: The string can remain empty, but must be included.*/
     string description;
};

/**
 * Return type for a command, also for an Operation starts a task.
 */
struct CommandReturnCode {

    ReturnState state;

    /** The system-wide identification of a task */
    XPERSIF::CommandTaskID commandTaskID;

    /** Description of the state of the task. Can contain information on errors.
     * optional: The string can remain empty, but must be included.      */
     string description;

    /** The forseen length to completion of the task
```

```
     * -1 : Unknown
     *  0 : Very short, not yet specified
     * >0 : Long duration
     */
long durationToFinish;
};

/*****************************************************************************************/
/**
* Functional operations as Commands of a component are contained in this interface.
* The operations defined here are architecturally prescribed and should be contained
* in component-specific interfaces by transmission.
*/
interface IOperation {
  /**
  * Start the component, initialize it and it is then ready to accept commands.
  *
  * @return ReturnCode
  */
  ReturnCode startComponent();

  /**
  * Breaks the component, interrupting the current task.
  *
  * @return ReturnCode
  */
  ReturnCode breakComponent();

  /**
  * Continues with the task the component was handling before breakComponent was called.
  *
  * @return ReturnCode
  */
  ReturnCode continueComponent();

  /**
  * Ends all running commands on the given component and sets the componentState to Standby.
  * The component can be restarted with StartComponent.
  *
  * @return ReturnCode
  */
  ReturnCode stopComponent();

  /**
  * Exits the component.
  *
  * @return ReturnCode
  */
  ReturnCode exitComponent();

  /**
   * Get the state of an issued command.
   *
   * @param commandID in: CommandTaskID datatype (created by the command which is called)
   * @param commandTaskInfo out: CommandTaskInfo datatype
```

```
     * @return ReturnCode
     */
    ReturnCode getCommandTaskState(XPERSIF::CommandTaskID commandTaskID,
    out XPERSIF::CommandTaskInfo commandTaskInfo);

    /**
     * State of a component.
     *
     * @param componentState out: The state of the component.
     * @return ComponentReturnCode
     */
    ReturnCode getComponentState (out XPERSIF::ComponentState componentState);

    /**
     * List of running commands.
     *
     * @param commands out: The running commands list.
     * @return ReturnCode
     */
    ReturnCode getRunningCommands(out CommandTaskIDlist commands);

    /**
     * Sets the component state.
     *
     * @param ccomponentState in: The new state of the component.
     * @return ReturnCode
     */
    ReturnCode setComponentState(XPERSIF::ComponentState componentState);
};

/*****************************************************************************************/
/**
* Functional operations as Commands of an organizational component are contained in this interface.
* The operations defined here are architecturally prescribed and should be contained
* in component-specific interfaces by transmission.
*/
struct ComponentStateInformation{
    XPERSIF::ComponentID id;
    XPERSIF::ComponentState state;
};

sequence<ComponentStateInformation> ComponentStateInformationList;

interface IOrganizationalOperation {
    /**
     * Registers the given component with the organizational component.
     *
     * @param componentID in: The ID of the component to be registered
     * @return ReturnCode
     */
    ReturnCode registerComponent(XPERSIF::ComponentID componentID);

    /**
     * Delivers a list of the registered components' state information.
     *
```

```
        * @param stateList out: The list fo registered components' state information
        * @return ReturnCode
        */
        ReturnCode getAllComponentStates(out ComponentStateInformationList stateList);

        /**
        * Delivers the state of a given component.
        *
        * @param componentID in: The ID of the component whose state will be delivered
        * @param state out: The state of the given component
        * @return ReturnCode
        */
        ReturnCode getStateOfComponent(XPERSIF::ComponentID componentID,
        out XPERSIF::ComponentState componentState);

        /**
        * Sets the state of a given component.
        *
        * @param componentID in: The ID of the component whose state will be set
        * @param state in: The new state of the given component
        * @return ReturnCode
        */
        ReturnCode setStateOfComponent(XPERSIF::ComponentID componentID,
        XPERSIF::ComponentState componentState);

        /**
        * Returns the CommandReturnCode when all components have reached the given state.
        * No commands to the components are issued to ensure they reach the state.
        * Their states are simply monitored and when all components are at the given state,
        * the command completes.
        *
        * @param componentState in: The state which is of the components
        * @return CommandReturnCode
        */
        CommandReturnCode monitorComponentState(XPERSIF::ComponentState componentState);

    };
};
#endif _XPERSIF_ICE
```

B.2 XPERSIFDT.ice

```
// File XPERSIFDT.ice
/**
* All communication between components is done in S.I. units (Systeme International d'Unites,
* International System of Units). This is also true for component configuration. Within the
* component code, S.I. units are used except within the hardware-specific skills and status class.
* In this case, the variable names and the documentation of the code must clearly specify the
* units.
*/

// Coordinate Frames:
// All angles are defined from -π to π (not 0 to 2π).

#ifndef _XPERSIFDT_ICE
```

```
#define _XPERSIFDT_ICE

module XPERSIFDT{

    /**
    * Xperiment Mode.
    */
    enum XperimentMode{
        simulated,
        physical
    };

    /**
    * Xecution Mode.
    */
    enum XecutionMode{
        online,
        offline
    };

/** LoopModel ***************************************************************************/

/** Manipulation ************************************************************************/

    /**
    * A 6D Pose.
    *
    * Pose = (x, y, z, roll, pitch, yaw)
    */
    struct Pose6D{
        double x;       // deviation in x-axis // [mm]
        double y;       // deviation in y-axis // [mm]
        double z;       // deviation in z-axis // [mm]
        double roll;    // rotation around x-axis // [rad]
        double pitch;   // rotation around y-axis // [rad]
        double yaw;     // rotation around z-axis // [rad]
    };

/** Relocation **************************************************************************/

    /**
    * A Pose in 2-dimensional space.
    *
    * Pose = (x, y, yaw)
    */
    struct Pose{
        double x;       // deviation in x-axis // [mm]
        double y;       // deviation in y-axis // [mm]
        double yaw; // rotation around z-axis // [rad]
    };

    /**
    * Description of paths as a sequence of 3D poses.
    */
    sequence<Pose> Path;
```

```
    /**
    * Tupel of velocity in 3-dimensional space.
    *
    * Velocity = (x, y, omega)
    */
    struct Velocity {
        double x;             // velocity in x-axis // [m/s]
        double y;             // velocity in y-axis // [m/s]
        double omega; // angular velocity       // [rad/sec]
    };

    /**
    * Pair of velocities for differential drive.
    *
    * DifferentialDriveVelocity = ( left , right )
    */
    struct DifferentialDriveVelocity {
        double left;     // velocity of left wheel // [m/s]
        double right;    // velocity of right wheel // [m/s]
    };

/** Perception ********************************************************************************/

    /**
    * enum specifying the various sensor types
    */
    enum SensorType{
        ir ,
        bump,
        sonar,
        camera,
        resistance,
        lightbarrier ,
        encoderPosition,
        encloderSpeed
    };

    /**
    * Struct specifying a sensor.
    */
    struct Sensor{
        SensorType type;    // Type of sensor
        int iD;             // system-wide unique identifier
        Pose6D pose;        // pose relative to the robot
    };

    /**
    * Struct specifying a sensor reading.
    */
    struct SensorReading{
        Sensor sensorID;            // sensor parameters
        double value;           // value of the sensor reading //What if its of a different type - int,etc??
        double accuracy;   //the accuracy with which we are certain the reading is correct
    };
```

```
    /**
    * Sequence of Sensor readings.
    */
    sequence<SensorReading> SensorReadingsVector;

    /**
    * Struct containing a sequences of sensor readings and the time they were read.
    */
    struct SensorReadings{
        double time;    // Time in seconds the readings were made.
        SensorReadingsVector vectorOfSensorReadings; // SensorReading vector
    };

/** XPERSimView ***************************************************************************/

    /**
    * Names of models currently used within the simulation
    */
    enum ModelIDs {
        unknown,
        myke,
        redSphere,
        wall1,
        wall2,
        wall3,
        wall4,
        redCube,
        yellowCube,
        blueCube,
        greenCube
    };

    /**
    * Sequence of NodeInfo.
    */
    sequence<ModelIDs> ModelIDsList;

    /**
    * Dictionary to map strings to ModelIDs.
    */
    dictionary<string, ModelIDs> ModelIDsMap;

    /**
    * Various types of Ogre Nodes
    */
    enum NodeType {
        modelNode,
        rayNode,
        lightNode,
        cameraNode,
        planeNode,
        skyBoxNode,
        skyDomeNode
    };
```

```
/**
 * A struct defining the orientation as a Quaternion = (w, x, y, z)
 */
struct ODEQuaternion {
    double w;
    double x;
    double y;
    double z;
};

/**
 * A struct defining the position (x, y, z) in the ODE coordinate system
 */
struct ODEPosition {
    double x;
    double y;
    double z;
};

/**
* A struct defining a simulated scene's node info
*/
struct NodeInfo{
    string name;
    ODEPosition position;
    ODEQuaternion orientation;
};

/**
* Sequence of NodeInfo.
*/
sequence<NodeInfo> NodeInfoList;

/**
* A struct defining a simulated image and its timestamp.
*/
struct ImageSim{
    double timestamp;
    NodeInfoList simScene;
};

/**
* A struct defining a simulated scene
*/
struct NodeConfig{
    NodeType type;
    string name;
    string parentNode;
    string mesh;
    string materialName;
    ODEPosition scale;
    ODEPosition position;
    ODEPosition orientation;
};
```

```
    /**
    * Sequence of NodeConfig.
    */
    sequence<NodeConfig> NodeConfigList;

/** Vision ************************************************************************************/

    /**
    * A struct specifying a 2D point
    */
    struct Point2D{
        int x;  // Position on screen in the horizontal  // [Pixel]
        int y;  // Position on screen in the vertical    // [Pixel]
    };

    /**
    * An enum of the possible color formats
    */
    enum ColorFormat{
        RGB,
        RGBa,
        HSV,
        YUV
    };

    /**
    * Sequence of color values with the specified color format.
    */
    sequence<int> ColorValues;

    /**
    * A struct specifying the color (of a model or an object)
    */
    struct Color{
        ColorFormat format;
        string colorName;
        ColorValues values;
    };

    /**
    * An enum of the possible shapes a model or an object may have
    */
    enum Shape{
        Sphere,
        Cube,
        Cylinder
    };

    /**
    * A struct specifying the dimentions of a model
    */
    struct Dimensions{
        double length;
        double width;
```

```
    double height;
    double radius;
};

/**
* An enum of the possible features to extract using the vision module
*/
enum VisionFeature{
    areaOnScreen,
    centerOnScreen
};

/**
* Sequence of values that may be the result of the feature extraction process.
*/
sequence<double> ValueList;

/**
* A struct specifying the Feature to be extracted, the parameters it takes and the
* result of the feature extraction process.
*/
struct VisionFeatureExtracted{
    VisionFeature feature;          // the feature to be extracted
    ValueList list ;                //the value(s) which was/were extracted
};

/**
* Sequence of FeaturesToExtract that may be sent to the feature extractor.
*/
sequence<VisionFeatureExtracted> VisionFeatureExtractedList;

/**
* Sequence of properties of an object which may include affordances.
*/
sequence<string> ObjectProperties;

/**
* A Model of an object that is provided either as a priori knowledge or has been
* previously recognized as a protoObject and then added to the KB for future use.
*
*/
struct Model{
    ModelIDs modelID;               // Unique identifier of the model
    string modelName;       // The name of the model (unique)
    Shape modelShape;       // The shape of the model
    ColorValues modelColor; // The color of the model /* in formatted output this is a string */
    Dimensions dim;         // The dimensions of the object depending of the shape
    Pose6D gripPose;        // The relative pose at which it is grippable
    ObjectProperties properties;  // Other properties of the object (affordances)
};

/**
* An object which has been recognized as being an instance of a model form the KB
*/
struct DetectedObject{
```

```
        int labelID;                // unique identifier of the instance of the model
        ModelIDs modelID;           // reference to the Model of which it is an instance
        Pose objectPose;    // 3D pose relative to the robot
        float accuracy;     // accuracy with which the detection was made (between 0.0 and 1.0)
        VisionFeatureExtractedList featureList; // may contain a list of features, their parameters and the value
    };

    /**
    * Sequence of Objects recognized.
    */
    sequence<DetectedObject> DetectedObjects;

    /**
    * A detected objects list with a timestamp.
    */
    struct DetectedObjectList
    {
        double time;
        DetectedObjects list;
    };

    /**
    * Sequence of Models
    */
    sequence<Model> ModelList;

    /**
    * A protoObject - a new object which is not an instance of any model within the KB.
    */
    // struct ProtoObject{
    /** TODO **/
    // }

/** Camera ***********************************************************************************/

    /**
    * An enum of the possible formats an image may have
    */
    enum ImageFormat{
        RGBValues,
        PNG
    };

    /**
    * A struct specifying the width and height of an image in pixels
    */
    struct ImageSize{
        int width;      // width of the image
        int height;     // height of the image
    };

    /**
    * An enum of the possible camera positions
    */
    enum CameraPosition{
```

```
        CAMTOP,   // overhead camera
        CAMLEFT,  // Left camera in stereo vision OR single camera
        CAMRIGHT // Right camera in stereo vision
    };

    /**
    * An **enum** of the possible types of processing    // Tricky! check this format
    */
    enum CameraTypeOfProcessing{
        STEREO, //0x01,
        TOP,          // 0x02,
        MONO        // 0x04
    };

    /**
    * Sequence of floats used to define a row.
    */
    sequence<float> Row;

    /**
    * Sequence of Rows used to define a matrix.
    */
    sequence<Row> Matrix;

    /**
    * A struct specifying the camera's information.
    */
    struct CameraInfo{
        int iD;        // system-wide unique identifier for the camera
        // string imageProvider;      // The image provider
        CameraPosition position;     // top, left , right
        Matrix intrinsicMatrix ;     // [3,3] camera intrinsic parameters
        Matrix extrinsicMatrix ;     // [4,4] camera extrinsic parameters
        Row distortions;                          // [4] camera distortion parameters
        ImageFormat cameraImageFormat; //e.g. RGB
        ImageSize cameraImageSize; // Size of the image from the camera in pixels
        CameraTypeOfProcessing typeOfProcessing; //stereo, top, Mono
    };

    /**
    * Sequence of cameras.
    */
    sequence<CameraInfo> CameraList;

    /**
    * A struct defining an image and its timestamp.
    */
    struct Image{
        double timestamp;
        //void *pData;  //This needs to be an Ice Stream
    };

/** FeatureExtraction Module ***********************************************************************/

    /**
```

```
 * An enum of the possible features to extract using the FeatureExtraction module
 */
enum Feature{
    currentTime,
    angleBetween,
    distanceBetween,
    poseOfObject,
    poseOfRobot,
    velocityOfObject,
    areaObjectOnScreen,
    centerObjectOnScreen,
    inContact,
    isObjectInView,
    isObjectBetweenGripper,
    actionsInExecution
};

/**
 * Sequence of objects that may be passed as paramerters for a feature extractor.
 */
sequence<ModelIDs> ParameterList;

/**
 * A struct specifying the Feature to be extracted and the parameters it takes.
 */
struct FeatureToExtract{
    int id;                                  // system-wide unique identifier of this instance
    Feature featureName;   // the feature to be extracted
    ParameterList parameters;   // the parameters needed for the feature
};

/**
 * Sequence of FeaturesToExtract that may be sent to the feature extractor.
 */
sequence<FeatureToExtract> FeatureList;

/**
 * A struct specifying the Logged Feature which has been extracted, the parameters it takes and the
 * result of the feature extraction process.
 */
struct ExtractedFeature{
    FeatureToExtract feature;          // the feature to be extracted
    ValueList featureValueList;  // the value of the extracted feature
};

/**
 * Sequence of features that make up the feature vector.
 */
sequence<ExtractedFeature> FeatureVector;

/**
 * Sequence of log vectors that make up a log trace.
 */
//sequence<LogVector> LogTrace;
```

```
/** Curiosity Module ***********************************************************************/

    /**
    * A struct specifying the curiosity index associated with a given pose
    */
    struct CuriosityNode{
        Pose nodePose;                  // The absolute coords of a node
        double curiosityIndex;  // The curiosity index attached to the node
    };

    /**
    * Sequence of CuriosityNodes forming a list/grid.
    */
    sequence<CuriosityNode> CuriosityNodeList;

    /**
    * A struct specifying the curiosity values and its timestamp.
    */
    struct CuriosityVector{
        int timeStep;
        CuriosityNodeList curiosityListOfNodes;
    };

/** Surprise Module ************************************************************************/
// These data types are taken mainly from current implementation of "Prediction-Design of experiments"

    // A string vector
    sequence<string> StringVector;

    /**
    * A struct specifying objects and their sensors
    */
    struct SObjects{
        string objectName;
        string sensorName;
    };

    /**
    * A struct specifying the objects and their sensors
    */
    sequence<SObjects> ObjectMapping;

    /**
    * A struct specifying the hypotheses and atoms that fail to explain a surprising event.
    */
    struct HypothesisInfo{
        string hypothesisName;
        StringVector relatedAtoms;
    };

    /**
    * Sequence of invalid hypotheses.
    */
    sequence<HypothesisInfo> InvalidHypotheses;
```

```
    /**
    * A struct specifying the all the surprise info and its timestamp.
    */
    struct SurpriseVector{
        int timeStep;
        InvalidHypotheses invalidHypothesesList;
        ObjectMapping objectMappingList;
    };

/** Motivation Module ***************************************************************************/

    /**
    * A struct specifying the different kinds of motivation and its timestamp.
    */
    struct MotivationVector{
        int timeStep;
        SurpriseVector surprise;
        CuriosityVector curiosity ;
    };

/** KnowledgeBase Module ************************************************************************/
// These data types are taken mainly from current implementation of "Prediction-Design of experiments"

    /**
    * An enum of the possible values used to describe a condition.
    */
    enum ConditionValue{
        change,
        trueValue,
        falseValue
    };

    /**
    * A struct specifying a sensor value and its condition value.
    */
    struct Condition{
        Sensor sensorOfCondition;
        ConditionValue valueOfcondition;
    };

    /**
    * Sequence of conditions.
    */
    sequence<Condition> ConditionList;

    /**
    * A struct specifying an Atomic sentence
    */
    struct Atom{
        string name;
        ModelList listOfModels;
        ConditionList listOfConditions ;
    };

    /**
```

```
    * A struct specifying a clause containing an atom and its boolean value
    */
    struct Clause{
        Atom atomInClause;
        bool value;
        bool atomic;
    };

    /**
    * Sequence of clauses.
    */
    sequence<Clause> ClauseList;

    /**
    * A struct specifying a hypothesis
    */
        struct Hypothesis{
        string hypothesisName;
        ClauseList listOfClauses;
    };

/** DesignOfExperiments Module ******************************************************************/

    /**
    * The various states the loop may be in
    */
    enum LoopState {
        DefaultState,
        SurpriseValidationState,
        ExperimentationState,
        LearnState
    };

    /**
    * Design concepts which may be passed to the Design components
    */
    enum DesignConcept {
        Default,
        SurpriseValidation,
        Experimentation
    };

/** Planning Module *****************************************************************************/

    /**
    * A struct specifying a goalConcept sent by a Design component
    */
    struct GoalConcept{
        string goalConceptName; //for now a string e.g. pushBoxes
    };

/** Execution Module ****************************************************************************/

    /**
    * Sequence of byte.
```

```
    */
    sequence<byte> PLEXILPlan;

};
#endif _XPERSIFDT_ICE
```

B.3 ApplicationCom.ice

```
// File ApplicationCom.ice
#ifndef _APPLICATIONCOM_ICE
#define _APPLICATIONCOM_ICE

#include "XPERSIFDT.ice"

module ApplicationCom{
    /**
     * ApplicationCom interface.
     */
    interface IApplicationCom {
        /**
         * Shuts down the application
         */
        idempotent void shutdown();
     };
};
#endif _APPLICATIONCOM_ICE
```

B.4 LoopModel.ice

```
// File LoopModel.ice
#ifndef _LOOPMODEL_ICE
#define _LOOPMODEL_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"

module LoopModel{

    /**
    * All Commands.
    */
    enum CommandType {
        monitorComponentState
    };

    /**
    * Observer of the LoopModel component.
    */
    interface IObserverOfLoopModel {
        void updateCalledByLoopModel(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * LoopModel as subject of other components which are observers of the
    * LoopModel component
    */
```

```
    interface ILoopModelSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfLoopModel* observerLoopModelRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfLoopModel* observerLoopModelRef);
    };

    interface ILoopModel extends XPERSIF::IOperation, XPERSIF::IOrganizationalOperation {
        /**
        * Sets the XperimentMode
        */
        void setXperimentMode(XPERSIFDT::XperimentMode xperimentMode);
    };

    /**
    * The LoopModel component and its interface.
    */
    interface ILoopModelComponent extends
        ILoopModel,
        ILoopModelSubject {
    };
};
#endif _LOOPMODEL_ICE
```

B.5 RobotModel.ice

```
// File RobotModel.ice
#ifndef _ROBOTMODEL_ICE
#define _ROBOTMODEL_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"

module RobotModel{

    /**
    * All Commands.
    */
    enum CommandType {
        monitorComponentState
    };

    /**
    * Observer of the RobotModel component.
    */
    interface IObserverOfRobotModel {
        void updateCalledByRobotModel(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * RobotModel as subject of other components which are observers of the
    * RobotModel component
    */
    interface IRobotModelSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfRobotModel *observerRobotModelRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfRobotModel *observerRobotModelRef);
    };
```

```
    interface IRobotModel extends XPERSIF::IOperation, XPERSIF::IOrganizationalOperation {
        /**
         * Sets the XperimentMode
           */
        void setXperimentMode(XPERSIFDT::XperimentMode xperimentMode);
        };

    /**
    * The RobotModel component and its interface.
    */
    interface IRobotModelComponent extends
        IRobotModel,
        IRobotModelSubject {
    };
};
#endif _ROBOTMODEL_ICE
```

B.6 Manipulation.ice

```
// File Manipulation.ice
#ifndef _MANIPULATION_ICE
#define _MANIPULATION_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"

module Manipulation{

    /**
    * All Commands.
    */
    enum CommandType {
        grip,
        ungrip,
        moveTooltipToPose,
        prepareToRelocate
    };

    /**
    * Observer for the Manipulation component.
    */
    interface IObserverOfManipulation {
        void updateCalledByManipulation(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * Manipulation as subject of other components which are observers of the
    * Manipulation component
    */
    interface IManipulationSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfManipulation *observerManipulationRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfManipulation *observerManipulationRef);
    };
```

```
/**
* Observer for Manipulation Raw Sensor Readings .
*/
interface IObserverOfManipulationRawSensorReadings {
    void setManipulationSensorReadings(XPERSIFDT::SensorReadings sensorReadings);
};

/**
* Manipulation as subject of other components which are raw data observers of the
* Manipulation component
*/
interface IManipulationRawSensorReadingsSubject {
    XPERSIF::ReturnCode attachRawSensorReadingsObserver(
    IObserverOfManipulationRawSensorReadings *observerManipulationRawSensorReadingsRef);
    XPERSIF::ReturnCode detachRawSensorReadingsObserver(
    IObserverOfManipulationRawSensorReadings *observerManipulationRawSensorReadingsRef);
};

interface IManipulation extends XPERSIF::IOperation {
    /**
    * Close the grippers until the object is griped.
    *
    * @Command
    * @precondition The grippers are open.
    * @precondition The object is grippable.
    * @precondition The object is between the grippers.
    * @effect The gripper is holding the object.
    *
    * @return CommandReturnCode
    */
    XPERSIF::CommandReturnCode grip();

    /**
    * Open the grippers to their maximum.
    *
    * @Command
    * @precondition The grippers are not in their maximum open position.
    * @effect The grippers are open to their maximum possible position.
    *
    * @return CommandReturnCode
    */
    XPERSIF::CommandReturnCode ungrip();

    /**
    * Moves the tooltip to the given pose (robot-centric coordinates) and refers
    * to the pose of which will be achieved if the gripper closes.
    *
    * @Command
    * @precondition The pose is reachable.
    * @effect The tooltip is in the given pose.
    *
    * @return CommandReturnCode
    */
    XPERSIF::CommandReturnCode moveTooltipToPose(XPERSIFDT::Pose6D pose);
```

```
        /**
        * Moves the manipulator to a pose that is safe during relocation
        *
        * @Command
        * @precondition The pose is reachable.
        * @effect The manipulator is in a safe pose.
        *
        * @return CommandReturnCode
        */
        XPERSIF::CommandReturnCode prepareToRelocate();

        /**
        * Delivers the pose of the tooltip in relative coordinates.
        *
        * @param distance out: The pose of the manipulator.
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getTooltipPose(out XPERSIFDT::Pose6D manipulatorPose);

        /**
        * Delivers the distance between the inner surfaces of the grippers.
        *
        * @param distance out: The distance between the two surfaces of the grippers in meters.
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getGripperPose(out double distance);

        /**
        * Delivers the presence of an object in between the grippers.
        *
        * @param objectPresent out: 1=object present; 0=no object present
        * @return ReturnCode
        */
        XPERSIF::ReturnCode isObjectPresent(out bool objectPresent);
    };

    /**
    * The Manipulation component and its interface.
    */
    interface IManipulationComponent extends
        IManipulation,
        IManipulationSubject,
        IManipulationRawSensorReadingsSubject {
    };
};
#endif _MANIPULATION_ICE
```

B.7 Relocation.ice

```
// File Relocation.ice
#ifndef _RELOCATION_ICE
#define _RELOCATION_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"
```

```
module Relocation{

    /**
    * All Commands.
    */
    enum CommandType {
        moveAbs,
        moveRel,
        followPathAbs,
        followPathRel,
        moveAbsInReverse,
        moveRelInReverse,
        moveForward,
        moveForwardAtSpeed,
        moveBackward,
        moveBackwardAtSpeed,
        rotate,
        rotateByAngle,
        stopDriving
    };

    /**
    * Observer of the Relocation component.
    */
    interface IObserverOfRelocation {
        void updateCalledByRelocation(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * Relocation as subject of other components which are observers of the
    * Relocation component
    */
    interface IRelocationSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfRelocation* observerRelocationRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfRelocation* observerRelocationRef);
    };

    /**
    * Observer of Relocation Raw Sensor Readings .
    */
    interface IObserverOfRelocationRawSensorReadings {
        void setRelocationSensorReadings(XPERSIFDT::SensorReadings sensorReadings);
    };

    /**
    * Relocation as subject of other components which are raw data observers of the
    * Relocation component
    */
    interface IRelocationRawSensorReadingsSubject {
        XPERSIF::ReturnCode attachRawSensorReadingsObserver(IObserverOfRelocationRawSensorReadings
            *observerRelocationRawSensorReadingsRef);
        XPERSIF::ReturnCode detachRawSensorReadingsObserver(IObserverOfRelocationRawSensorReadings *
            observerRelocationRawSensorReadingsRef);
    };
```

```
interface IRelocation extends XPERSIF::IOperation {
/**
* Move the robot to an absolute position.
*
* @Command
* @precondition The robot is movable.
* @precondition The velocities are attainable.
* @effect The robot is in the given pose.
*
* @param pose in: The pose in absolute coordinates
* @return CommandReturnCode
*/
XPERSIF::CommandReturnCode moveAbs(XPERSIFDT::Pose pose);

/**
* Drive to a relative Position.
*
* @Command
* @precondition The robot is movable.
* @precondition The pose is attainable.
* @effect The robot is in the Position currentPose+deltaPose.
*
* @param deltaPose in: The relative Position
* @return CommandReturnCode
*/
XPERSIF::CommandReturnCode moveRel(XPERSIFDT::Pose deltaPose);

/**
* Moves the robot along a specified path given in absolute coordinates.
*
* @param path in: Path as a sequence of absolute poses
* @return CommandReturnCode
*/
XPERSIF::CommandReturnCode followPathAbs(XPERSIFDT::Path path);

/**
* Moves the robot along a specified path.
*
* @param path in: Path as a sequence of relative poses
* @return CommandReturnCode
*/
XPERSIF::CommandReturnCode followPathRel(XPERSIFDT::Path path);

/**
* Move the robot to an absolute position while driving in reverse.
*
* @Command
* @precondition The robot is movable.
* @precondition The velocities are attainable.
* @effect The robot is in the given pose.
*
* @param pose in: The pose in absolute coordinates
* @return CommandReturnCode
*/
```

```
XPERSIF::CommandReturnCode moveAbsInReverse(XPERSIFDT::Pose pose);

/**
* Drive to a relative Position while driving in reverse.
*
* @Command
* @precondition The robot is movable.
* @precondition The pose is attainable.
* @effect The robot is in the Position currentPose+deltaPose.
*
* @param deltaPose in: The relative Position
* @return CommandReturnCode
*/
XPERSIF::CommandReturnCode moveRelInReverse(XPERSIFDT::Pose deltaPose);

/**
* Moves the robot forward at the minimum moving speed.
*
* @return CommandReturnCode
*/
XPERSIF::CommandReturnCode moveForward();

/**
* Moves the robot forward at the given speed.
*
* @param speed: The linear speed the robot should move at
* @return CommandReturnCode
*/
XPERSIF::CommandReturnCode moveForwardAtSpeed(double speed);

/**
* Moves the robot backward at the minimum moving speed.
*
* @return CommandReturnCode
*/
XPERSIF::CommandReturnCode moveBackward();

/**
* Moves the robot backward at the given speed.
*
* @param speed: The linear speed the robot should move at
* @return CommandReturnCode
*/
XPERSIF::CommandReturnCode moveBackwardAtSpeed(double speed);

/**
* Rotates the robot to the given relative angle.
*
* @param angle in: Angle in radians: positive value--left and negative value--right
* @return CommandReturnCode
*/
XPERSIF::CommandReturnCode rotateByAngle(double angle);

/**
* Rotates the robot at the minimum moving speed.
```

```
        *
        * @return CommandReturnCode
        */
        XPERSIF::CommandReturnCode rotate();

        /**
        * Sets the desired speed to 0 in order to stop driving.
        * @command
        * @precondition The robot has a non-zero velocity
        * @effect The robot is not driving
        * @return CommandReturnCode
        */
        XPERSIF::CommandReturnCode stopDriving();

        /**
        * Move the robot with the given velocity and direction.
        *
        * @precondition The velocity lies within a specific range of values
        * @effect The given velocity is used by the next command
        *
        * @param velocity in: Velocity struct
        * @return ReturnCode
        */
        XPERSIF::ReturnCode setMaximumRobotVelocities(XPERSIFDT::Velocity velocity);

        /**
        * Set the accuracy with which the pose will be evaluated
        *
        * @effect The accuracy is set.
        *
        * @param accuracyPosition in: The accuracy of the position
        * @param accuracyYaw in: The accuracy of the yaw orientation
        * @return ReturnCode
        */
        XPERSIF::ReturnCode setAccuracy(double accuracyPosition, double accuracyYaw);

        /**
        * Delivers the speeds of the robot.
        *
        * @param velocity out: Velocity struct
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getCurrentSpeed(out XPERSIFDT::Velocity velocity);

        /**
        * Delivers the position of the robot.
        *
        * @param currentPose out: The current pose of the robot
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getCurrentPose(out XPERSIFDT::Pose currentPose);
    };

    /**
    * The Relocation component and its interface.
```

```
    */
    interface IRelocationComponent extends
        IRelocation,
        IRelocationSubject,
        IRelocationRawSensorReadingsSubject {
    };
};
#endif _RELOCATION_ICE
```

B.8 Perception.ice

```
// File Perception.ice
#ifndef _PERCEPTION_ICE
#define _PERCEPTION_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"
#include "XPERSimView.ice"
#include "Camera.ice"
#include "RobotFeatureExtraction.ice"

module Perception{
    /**
    * All Commands.
    */
    enum CommandType {
        monitorComponentState
    };

    /**
    * Observer of the Perception component.
    */
    interface IObserverOfPerception {
        void updateCalledByPerception(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * Perception as subject of other components which are observers of the
    * Perception component
    */
    interface IPerceptionSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfPerception *observerPerceptionRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfPerception *observerPerceptionRef);
    };

    /**
    * Observer for Perception Raw Sensor Readings .
    */
    interface IObserverOfPerceptionRawSensorReadings {
        void setPerceptionSensorReadings(XPERSIFDT::SensorReadings sensorReadings);
    };

    /**
    * Perception as subject of other components which are raw data observers of the
    * Perception component
```

```
*/
interface IPerceptionRawSensorReadingsSubject {
    XPERSIF::ReturnCode attachRawSensorReadingsObserver(IObserverOfPerceptionRawSensorReadings
                                    *observerPerceptionRawSensorReadingsRef);
    XPERSIF::ReturnCode detachRawSensorReadingsObserver(IObserverOfPerceptionRawSensorReadings
                                    *observerPerceptionRawSensorReadingsRef);
};

interface IPerception extends XPERSIF::IOperation, XPERSIF::IOrganizationalOperation {
    /**
    * Delivers a list of all the objects which have been detected within a given image frame.
    *
    * @param objectList out: The list of detected objects
    * @return ReturnCode
    */
    XPERSIF::ReturnCode getObjectsInView(out XPERSIFDT::DetectedObjectList detectedObjectList);

    /**
    * Delivers the pose of an object
    *
    * @param objectLabelID in: The object whose pose should be delivered
    * @param pose out: The pose of the object whose identification was passed
    * @return ReturnCode
    */
    XPERSIF::ReturnCode getObjectPose(XPERSIFDT::ModelIDs objectLabelID, out XPERSIFDT::Pose pose);

    /**
    * Delivers a boolean value specifying whether a given object is within the field of view.
    *
    * @param objectLabelID in: The object whose presence should be delivered.
    * @param bool out: The value specifying the presence of the object.
    * @return ReturnCode
    */
    XPERSIF::ReturnCode isObjectInView(XPERSIFDT::ModelIDs objectLabelID, out bool inView);

    /**
    * Delivers the percentage of pixels on screen that a redSphere occupies.
    *
    * @param objectLabelID in: The object whose area on screen should be delivered.
    * @param double out: The object whose area on screen should be delivered.
    * @return ReturnCode
    */
    XPERSIF::ReturnCode getObjectArea(XPERSIFDT::ModelIDs objectLabelID, out double area);

    /**
    * Delivers the type of processing of the cameras (stereo,top,mono).
    *
    * @param typeOfProcessing out: The type of processing.
    * @return ReturnCode
    */
    XPERSIF::ReturnCode getObjectCenterOnScreen(XPERSIFDT::ModelIDs objectLabelID,
        out XPERSIFDT::Point2D centerOnScreen);

    /**
    * Delivers the type of processing of the cameras (stereo,top,mono).
```

```
*
* @param typeOfProcessing out: The type of processing.
* @return ReturnCode
*/
XPERSIF::ReturnCode getTypeOfProcessing(out XPERSIFDT::CameraTypeOfProcessing typeOfProcessing);

/**
* Delivers the list of available Cameras.
*
* @param cameraList out: The list of available cameras.
* @return ReturnCode
*/
XPERSIF::ReturnCode getCameras(out XPERSIFDT::CameraList cameraList);

/**
* Sets the list of detected objects within an image frame
*
* @param time in: The time stamp of the image frame in which the objects were detected.
* @param detectedObjectList in: The list of detected objects.
* @return ReturnCode
*/
XPERSIF::ReturnCode setDetectedObjects(double time,
    XPERSIFDT::DetectedObjectList detectedObjectList);

/**
* Delivers the distance between two objects
*
* @param objectLabelID1 in: A string that identifies the first object
* @param objectLabelID2 in: A string that identifies the second object
* @param distance out: The distance between the objects whose identifications was passed
* @return ReturnCode
*/
XPERSIF::ReturnCode getDistanceBetweenObjects(XPERSIFDT::ModelIDs objectLabelID1, XPERSIFDT::
    ModelIDs objectLabelID2, out double distance);

/**
* Delivers the angle between two objects
*
* @param objectLabelID1 in: the first object
* @param objectLabelID2 in: the second object
* @param distance out: The angle between the objects whose identifications was passed
* @return ReturnCode
*/
XPERSIF::ReturnCode getAngleBetweenObjects(XPERSIFDT::ModelIDs objectLabelID1, XPERSIFDT::
    ModelIDs objectLabelID2, out double angle);

/**
* Delivers the IR Sensor Readings.
*
* @param IRreadings out: The sensor readings from an IR sensor
* @return ReturnCode
*/
XPERSIF::ReturnCode getIR(out XPERSIFDT::SensorReadingsVector IRreadings);

/**
```

```
        * Delivers the Bumper Sensor Readings.
        *
        * @param bumpSensorReadings out: The sensor readings from an bumper sensor
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getBumper(out XPERSIFDT::SensorReadings bumpSensorReadings);

        /**
        * Delivers the time
        *
        * @param time out: The time in ms since the start of an experiment
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getTime(out double time);
    };

    /**
    * The Perception component and its interface.
    */
    interface IPerceptionComponent extends
        IPerception,
        IPerceptionSubject,
        IPerceptionRawSensorReadingsSubject,
        XPERSimView::IXPERSimView,
        Camera::IImageSubject,
        RobotFeatureExtraction::IObserverOfRobotFeatureExtraction
    {
    };
};
#endif _PERCEPTION_ICE
```

B.9 Observation.ice

```
// File Observation.ice
#ifndef _OBSERVATION_ICE
#define _OBSERVATION_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"
#include "XPERSimView.ice"
#include "Camera.ice"
#include "RobotFeatureExtraction.ice"

module Observation{

    /**
    * All Commands.
    */
    enum CommandType {
        monitorComponentState
    };

    /**
    * Observer of the Observation component.
    */
```

```
interface IObserverOfObservation {
    void updateCalledByObservation(XPERSIF::CommandTaskInfo commandTaskInfo);
};

/**
* Observation as subject of other components which are observers of the
* Observation component
*/
interface IObservationSubject {
    XPERSIF::ReturnCode attachObserver(IObserverOfObservation *observerObservationRef);
    XPERSIF::ReturnCode detachObserver(IObserverOfObservation *observerObservationRef);
};

interface IObservation extends XPERSIF::IOperation, XPERSIF::IOrganizationalOperation {
    /**
    * Sets the XperimentMode
    */
    void setXperimentMode(XPERSIFDT::XperimentMode xperimentMode);

    /**
    * Delivers a list of all the objects which have been detected within a given image frame.
    *
    * @param objectList out: The list of detected objects
    * @return ReturnCode
    */
    XPERSIF::ReturnCode getObjectsInView(out XPERSIFDT::ModelIDsList detectedObjectList);

    /**
    * Delivers the pose of an object
    *
    * @param objectLabelID in: The object whose pose should be delivered
    * @param pose out: The pose of the object whose identification was passed
    * @return ReturnCode
    */
    XPERSIF::ReturnCode getObjectPose(XPERSIFDT::ModelIDs objectLabelID, out XPERSIFDT::Pose pose);

    /**
    * Delivers the robot pose as extracted from the overhead camera.
    *
    * @param pose out: The robot pose
    * @return ReturnCode
    */
    XPERSIF::ReturnCode getRobotPose(out XPERSIFDT::Pose robotPose);

    /**
    * Delivers a boolean value specifying whether a given object is within the field of view.
    *
    * @param objectLabelID in: The object whose presence should be delivered.
    * @param bool out: The value specifying the presence of the object.
    * @return ReturnCode
    */
    XPERSIF::ReturnCode isObjectInView(XPERSIFDT::ModelIDs objectLabelID, out bool inView);

    /**
    * Delivers the type of processing of the cameras (stereo,top,mono).
```

```
        *
        * @param typeOfProcessing out: The type of processing.
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getTypeOfProcessing(out XPERSIFDT::CameraTypeOfProcessing typeOfProcessing);

        /**
        * Delivers the list of available Cameras.
        *
        * @param cameraList out: The list of available cameras.
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getCameras(out XPERSIFDT::CameraList cameraList);

        /**
        * Delivers the distance between two objects
        *
        * @param objectLabelID1 in: A string that identifies the first object
        * @param objectLabelID2 in: A string that identifies the second object
        * @param distance out: The distance between the objects whose identifications was passed
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getDistanceBetweenObjects(XPERSIFDT::ModelIDs objectLabelID1, XPERSIFDT::
            ModelIDs objectLabelID2,
              out double distance);

        /**
        * Delivers the angle between two objects
        *
        * @param objectLabelID1 in: the first object
        * @param objectLabelID2 in: the second object
        * @param distance out: The angle between the objects whose identifications was passed
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getAngleBetweenObjects(XPERSIFDT::ModelIDs objectLabelID1, XPERSIFDT::
            ModelIDs objectLabelID2, out double angle);

        /**
        * Delivers the time
        *
        * @param time out: The time in ms since the start of an experiment
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getTime(out double time);
    };

    /**
    * The Observation component and its interface.
    */
    interface IObservationComponent extends
        IObservation,
        IObservationSubject,
        XPERSimView::IXPERSimView,
        Camera::IImageSubject,
        RobotFeatureExtraction::IObserverOfRobotFeatureExtraction
```

```
    {
    };
};
#endif _OBSERVATION_ICE
```

B.10 Camera.ice

```
// File Camera.ice
// Needs to implement XPERSimView::IImageSimObserver, TeleView::IImageObserver
#ifndef _CAMERA_ICE
#define _CAMERA_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"
#include "XPERSimView.ice"

module Camera{
    /**
    * All Commands.
    */
    //enum CommandType {
    //};

    /**
    * Observer of an image (real camera)
    */
    interface IImageObserver {
        void setFrame(XPERSIFDT::Image image);
    };

    /**
    * The image as the Subject of its observers (Real camera).
    */
    interface IImageSubject {
        XPERSIF::ReturnCode attachImageObserver(IImageObserver* imageRef);
        XPERSIF::ReturnCode detachImageObserver(IImageObserver* imageRef);
    };

    interface ICamera {
        /**
        * Sets the XperimentMode
        */
        void setXperimentMode(XPERSIFDT::XperimentMode xperimentMode);

        /**
        * Delivers the information of the camera.
        *
        * @param cameraInfo out: The information of the camera
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getCameraInfo(out XPERSIFDT::CameraInfo cameraInfo);

        /**
        * Delivers the position of the camera (top, left or right).
        *
```

```
     * @param cameraPosition out: The position of the camera
     * @return ReturnCode
     */
    XPERSIF::ReturnCode getCameraPosition(out XPERSIFDT::CameraPosition cameraPosition);

    /**
     * Sets the id of the camera.
     *
     * @param cameraID in: The ID of the camera
     * @return ReturnCode
     */
    XPERSIF::ReturnCode setCameraId(int cameraID);

    /**
     * Sets the matrix of the camera's intrinsic parameters.
     *
     * @param intrinsicMatrix in: The intrinsic parameters' matrix
     * @return ReturnCode
     */
    XPERSIF::ReturnCode setIntrinsicParameters(XPERSIFDT::Matrix intrinsicMatrix);

    /**
     * Sets the matrix of the camera's extrinsic parameters.
     *
     * @param extrinsicMatrix in: The extrinsic parameters' matrix
     * @return ReturnCode
     */
    XPERSIF::ReturnCode setExtrinsicParameters(XPERSIFDT::Matrix extrinsicMatrix);

    /**
     * Sets the matrix of the camera's distortion parameters.
     *
     * @param distortions in: The distortion parameters' unidimentional matrix
     * @return ReturnCode
     */
    XPERSIF::ReturnCode setDistortionParameters(XPERSIFDT::Row distortions);

    /**
     * Sets the format of the image (e.g. PNG)
     *
     * @param imageFormat in: The format of the image
     * @return ReturnCode
     */
    XPERSIF::ReturnCode setImageFormat(XPERSIFDT::ImageFormat imageFormat);

    /**
     * Sets the size of the image (width,height)
     *
     * @param imageSize in: The size of the image
     * @return ReturnCode
     */
    XPERSIF::ReturnCode setImageSize(XPERSIFDT::ImageSize imageSize);

    /**
     * Sets the position of the camera (top, left or right).
```

```
        *
        * @param cameraPosition in: The position of the camera
        * @return ReturnCode
        */
        XPERSIF::ReturnCode setCameraPosition(XPERSIFDT::CameraPosition cameraPosition);

        /**
        * Sets the type of processing of the camera (stereo, top or mono).
        *
        * @param cameraTypeOfProcessing in: The type of processing of the camera
        * @return ReturnCode
        */
        XPERSIF::ReturnCode setCameraTypeOfProcessing(
            XPERSIFDT::CameraTypeOfProcessing cameraTypeOfProcessing);

        /**
        * Ask the camera component to connect with the camera (simulated or real)
        *
        * @return ReturnCode
        */
        XPERSIF::ReturnCode connect();

        /**
        * Ask the camera component to disconnect with the camera (simulated or real)
        *
        * @return ReturnCode
        */
        XPERSIF::ReturnCode disconnect();
    };

    /**
    * The Camera component and its interface.
    */
    interface ICameraComponent extends
        ICamera, //Operations above
        IImageSubject, //allow subscriptions to real image
        XPERSimView::IXPERSimView, //allow subscriptions to simulated image and fetching the scene
        XPERSimView::IImageSimObserver //allow camera to act as a teleobserver and receive updates from xpersim
    {
    };
};
#endif _CAMERA_ICE
```

B.11 GoalDesign.ice

```
// File GoalDesign.ice
// Needs to implement FeatureExtraction::IObserverOfFeatureVector,Motivation::IObserverOfMotivationVector,
// Needs to observe Planning, Execution
#ifndef _GOALDESIGN_ICE
#define _GOALDESIGN_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"
#include "Planning.ice"
#include "FeatureExtraction.ice"
```

```
#include "Motivation.ice"

module GoalDesign{
    /**
    * All Commands.
    */
    enum CommandType {
        generateGoalConcept
    };

    /**
    * Observer of the GoalDesign component.
    */
    interface IObserverOfGoalDesign {
        void updateCalledByGoalDesign(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * GoalDesign as subject of other components which are observers of the
    * GoalDesign component
    */
    interface IGoalDesignSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfGoalDesign *observerGoalDesignRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfGoalDesign *observerGoalDesignRef);
    };

    interface IGoalDesign extends XPERSIF::IOperation {
        /**
        * Returns the responsibility of orchestrating the loop to GoalDesign.
        *
        * @Operation
        * @precondition None.
        * @effect GoalDesign takes back the responsibility for orchestrating the loop
        *
        * @return CommandReturnCode
        */
        XPERSIF::ReturnCode takeControl();

    };

    /**
    * The GoalDesign component and its interface.
    */
    interface IGoalDesignComponent extends
        IGoalDesign,
        IGoalDesignSubject,
        FeatureExtraction::IObserverOfFeatureVector,
        Motivation::IObserverOfMotivationVector,
        Planning::IObserverOfPlans
    {
    };
};
#endif _GOALDESIGN_ICE
```

B.12 DesignOfExperiments.ice

```
// File DesignOfExperiments.ice
// Needs to implement FeatureExtraction::IObserverOfFeatureVector,Motivation::IObserverOfMotivationVector
#ifndef _DESIGNOFEXPERIMENTS_ICE
#define _DESIGNOFEXPERIMENTS_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"
#include "Planning.ice"
#include "FeatureExtraction.ice"
#include "Motivation.ice"

module DesignOfExperiments{
    /**
    * All Commands.
    */
    enum CommandType {
        generateGoalConcept
    };

    /**
    * Observer of the DesignOfExperiments component.
    */
    interface IObserverOfDesignOfExperiments {
        void updateCalledByDesignOfExperiments(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * DesignOfExperiments as subject of other components which are observers of the
    * DesignOfExperiments component
    */
    interface IDesignOfExperimentsSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfDesignOfExperiments *observerDesignOfExperimentsRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfDesignOfExperiments *observerDesignOfExperimentsRef)
            ;
    };

    interface IDesignOfExperiments extends XPERSIF::IOperation {
        /**
        * Generate a goal concept for the Planner.
        *
        * @Command
        * @precondition Motivation and feature vectors have been received.
        * @effect A goal concept is generated
        *
        * @param designConcept in: One of the states for what a goal concept may be generated
        * @return CommandReturnCode
        */
        XPERSIF::CommandReturnCode designExperiment();
    };

    /**
    * The DesignOfExperiments component and its interface.
    */
    interface IDesignOfExperimentsComponent extends
        IDesignOfExperiments,
```

```
        IDesignOfExperimentsSubject,
        FeatureExtraction::IObserverOfFeatureVector,
        Motivation::IObserverOfMotivationVector,
        Planning::IObserverOfPlans
    {
    };
};
#endif _DESIGNOFEXPERIMENTS_ICE
```

B.13 Planning.ice

```
// File Planning.ice
#ifndef _PLANNING_ICE
#define _PLANNING_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"

module Planning{
    /**
    * All Commands.
    */
    enum CommandType {
        generatePlan
    };

    /**
    * Observer of the Planning component.
    */
    interface IObserverOfPlanning {
        void updateCalledByPlanning(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * Planning as subject of other components which are observers of the
    * Planning component
    */
    interface IPlanningSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfPlanning* observerPlanningRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfPlanning* observerPlanningRef);
    };

    /**
    * Observer of plans.
    */
    interface IObserverOfPlans {
        void setPlan(XPERSIFDT::PLEXILPlan plan);
    };

    interface IPlanning extends XPERSIF::IOperation {
        /**
        * Generates a plan to archive a given goalConcept and encodes it in Plaxil lenguage.
        *
        * @Command
        * @precondition The goalConcept is achievable.
```

```
        * @effect A plexil encoded plan is generated.
        *
        * @param goalConcept in: The goal concept
        * @return CommandReturnCode
        */
        XPERSIF::CommandReturnCode generatePlan(XPERSIFDT::GoalConcept goalConcept, IObserverOfPlans*
            observerOfPlansRef );
    };

    /**
    * The Planning component and its interface.
    */
    interface IPlanningComponent extends
        IPlanning,
        IPlanningSubject    {
    };
};
#endif _PLANNING_ICE
```

B.14 Execution.ice

```
// File Execution.ice
#ifndef _EXECUTION_ICE
#define _EXECUTION_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"

module Execution{

    /**
     * All Commands.
     */
    enum CommandType {
        executePlan
    };

    /**
    * Observer for the Execution component.
    */
    interface IObserverOfExecution {
        void updateCalledByExecution(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * Execution as subject of other components which are observers of the
    * Execution component
    */
    interface IExecutionSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfExecution* observerExecutionRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfExecution* observerExecutionRef);
    };

    interface IExecution extends XPERSIF::IOperation {
        /**
```

```
        * Executes a given PLEXIL plan.
        *
        * @Command
        * @precondition The robotic embodiment is in a ready state.
        * @effect The plan is executed.
        *
        * @param PLEXILPlan in: The PLEXIL encoded plan to be executed
        * @return CommandReturnCode
        */
        // XPERSIF::CommandReturnCode executePlan(XPERSIFDT::PLEXILPlan PLEXILPlan);
        XPERSIF::CommandReturnCode executePlan();
    };

    /**
    * The Execution component and its interface.
    */
    interface IExecutionComponent extends
        IExecution,
        IExecutionSubject   {
    };
};
#endif _EXECUTION_ICE
```

B.15 FeatureSelection.ice

```
// File FeatureSelection.ice
#ifndef _FEATURESELECTION_ICE
#define _FEATURESELECTION_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"

module FeatureSelection{

    /**
    * All Commands.
    */
    // enum CommandType {
    // };

    /**
    * Observer for the FeatureSelection component.
    */
    interface IObserverOfFeatureSelection {
        void updateCalledByFeatureSelection(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * FeatureSelection as subject of other components which are observers of the
    * FeatureSelection component
    */
    interface IFeatureSelectionSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfFeatureSelection* observerFeatureSelectionRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfFeatureSelection* observerFeatureSelectionRef);
    };
```

```
    interface IFeatureSelection extends XPERSIF::IOperation {
    };

    /**
    * The FeatureSelection component and its interface.
    */
    interface IFeatureSelectionComponent extends
        IFeatureSelection,
        IFeatureSelectionSubject    {
    };
};
#endif _FEATURESELECTION_ICE
```

B.16 FeatureExtraction.ice

```
// File FeatureExtraction.ice
// Needs to implement Vision::IObserverOfVisionFeatureVector
#ifndef _FEATUREEXTRACTION_ICE
#define _FEATUREEXTRACTION_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"

module FeatureExtraction{
    /**
    * All Commands.
    */
    enum CommandType {
        monitorComponentState
    };

    /**
    * Observer for the FeatureExtraction component.
    */
    interface IObserverOfFeatureExtraction {
        void updateCalledByFeatureExtraction(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * FeatureExtraction as subject of other components which are observers of the
    * FeatureExtraction component
    */
    interface IFeatureExtractionSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfFeatureExtraction* observerFeatureExtractionRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfFeatureExtraction* observerFeatureExtractionRef);
    };

    /**
    * Observer of the FeatureVector
    */
    interface IObserverOfFeatureVector {
        void setFeatureVector(XPERSIFDT::FeatureVector featureVector);
    };
```

```
    /**
    * FeatureVector as subject of other components which are observers of the
    * FeatureVector
    */
    interface IFeatureVectorSubject {
        XPERSIF::ReturnCode attachFeatureObserver(IObserverOfFeatureVector* observerFeatureVectorRef);
        XPERSIF::ReturnCode detachFeatureObserver(IObserverOfFeatureVector* observerFeatureVectorRef);
    };

    interface IFeatureExtraction extends XPERSIF::IOperation, XPERSIF::IOrganizationalOperation {
        /**
        * Sets the XperimentMode
        */
        void setXperimentMode(XPERSIFDT::XperimentMode xperimentMode);

        /**
        * Set a list of features to extract by the FeatureExtraction Component.
        *
        * @param featureList in: The list of features to extract
        * @return ReturnCode
        */
        XPERSIF::ReturnCode setFeaturesToExtract(XPERSIFDT::FeatureList featureList);

        /**
        * Add a new feature to extract by the FeatureExtraction Component.
        *
        * @param featureToExtract in: The feature to extract
        * @return ReturnCode
        */
        XPERSIF::ReturnCode addFeatureToExtract(XPERSIFDT::FeatureToExtract featureToExtract);

        /**
        * Remove a feature from the list of features to extract by the FeatureExtraction Component.
        *
        * @param featureToExtract in: The feature to remove
        * @return ReturnCode
        */
        XPERSIF::ReturnCode removeFeatureToExtract(XPERSIFDT::FeatureToExtract featureToExtract);
    };

    /**
    * The FeatureExtraction component and its interface.
    */
    interface IFeatureExtractionComponent extends
        IFeatureExtraction,
        IFeatureExtractionSubject,
        IFeatureVectorSubject {
    };
};
#endif _FEATUREEXTRACTION_ICE
```

B.17 RobotFeatureExtraction.ice

```
// File RobotFeatureExtraction.ice
// Needs to implement Vision::IObserverOfVisionFeatureVector
```

```
#ifndef _ROBOTFEATUREEXTRACTION_ICE
#define _ROBOTFEATUREEXTRACTION_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"
#include "Vision.ice"

module RobotFeatureExtraction{
    /**
    * All Commands.
    */
    enum CommandType {
        monitorComponentState
    };

    /**
    * Observer for the RobotFeatureExtraction component.
    */
    interface IObserverOfRobotFeatureExtraction {
        void updateCalledByRobotFeatureExtraction(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * RobotFeatureExtraction as subject of other components which are observers of the
    * RobotFeatureExtraction component
    */
    interface IRobotFeatureExtractionSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfRobotFeatureExtraction*
            observerRobotFeatureExtractionRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfRobotFeatureExtraction*
            observerRobotFeatureExtractionRef);
    };

    interface IRobotFeatureExtraction extends XPERSIF::IOperation, XPERSIF::IOrganizationalOperation {

        /**
        * Sets the XperimentMode
        */
        void setXperimentMode(XPERSIFDT::XperimentMode xperimentMode);

        /**
        * Delivers a list of all the objects which have been detected within a given image frame.
        *
        * @param objectList out: The list of detected objects
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getObjectsInView(out XPERSIFDT::ModelIDsList detectedObjectList);

        /**
        * Delivers the pose of an object
        *
        * @param objectLabelID in: The object whose pose should be delivered
        * @param pose out: The pose of the object whose identification was passed
        * @return ReturnCode
        */
```

```
        XPERSIF::ReturnCode getObjectPose(XPERSIFDT::ModelIDs objectLabelID, out XPERSIFDT::Pose pose);

        /**
        * Delivers the robot pose as extracted from the overhead camera.
        *
        * @param pose out: The robot pose
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getRobotPose(out XPERSIFDT::Pose robotPose);

        /**
        * Delivers a boolean value specifying whether a given object is within the field of view.
        *
        * @param objectLabelID in: The object whose presence should be delivered.
        * @param bool out: The value specifying the presence of the object.
        * @return ReturnCode
        */
        XPERSIF::ReturnCode isObjectInView(XPERSIFDT::ModelIDs objectLabelID, out bool inView);

        /**
        * Delivers the distance between two objects
        *
        * @param objectLabelID1 in: A string that identifies the first object
        * @param objectLabelID2 in: A string that identifies the second object
        * @param distance out: The distance between the objects whose identifications was passed
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getDistanceBetweenObjects(XPERSIFDT::ModelIDs objectLabelID1, XPERSIFDT::
            ModelIDs objectLabelID2,
              out double distance);

        /**
        * Delivers the angle between two objects
        *
        * @param objectLabelID1 in: the first object
        * @param objectLabelID2 in: the second object
        * @param distance out: The angle between the objects whose identifications was passed
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getAngleBetweenObjects(XPERSIFDT::ModelIDs objectLabelID1, XPERSIFDT::
            ModelIDs objectLabelID2, out double angle);

        /**
        * Delivers the time
        *
        * @param time out: The time in ms since the start of an experiment
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getTime( out double time);

    };

    /**
    * The RobotFeatureExtraction component and its interface.
    */
```

```
    interface IRobotFeatureExtractionComponent extends
        IRobotFeatureExtraction,                // IOperation + its own commands and operations
        IRobotFeatureExtractionSubject, //subject of subscribers
        Vision::IObserverOfVisionFeatureVector //Allows it to receive low-level features from vision
    {
    };
};
#endif _ROBOTFEATUREEXTRACTION_ICE
```

B.18 Vision.ice

```
// File Vision.ice
#ifndef _VISION_ICE
#define _VISION_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"

module Vision{
    /**
    * All Commands.
    */
    // enum CommandType {
    // };

    /**
    * Observer for the Vision component.
    */
    interface IObserverOfVision {
        void updateCalledByVision(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * Vision as subject of other components which are observers of the
    * Vision component
    */
    interface IVisionSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfVision* observerVisionRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfVision* observerVisionRef);
    };

    /**
    * Observer for the Vision Feature Vector.
    */
    interface IObserverOfVisionFeatureVector {
        void setVisionFeatureVector(XPERSIFDT::DetectedObjectList detectedObjectList);
    };

    /**
    * Vision as subject of other components which are observers of the
    * Vision feature vector
    */
    interface IVisionFeatureVectorSubject {
        XPERSIF::ReturnCode attachVisionFeatureObserver(IObserverOfVisionFeatureVector
                                        *observerVisionFeatureVectorRef);
```

```
        XPERSIF::ReturnCode detachVisionFeatureObserver(IObserverOfVisionFeatureVector
                                                *observerVisionFeatureVectorRef);
    };

    interface IVision extends XPERSIF::IOperation {
        /**
        * Sets the XperimentMode
        */
        void setXperimentMode(XPERSIFDT::XperimentMode xperimentMode);

        /**
        * Add a new feature to extract by the Vision Component.
        *
        * @param visionfeature in: The vision feature to extract
        * @return ReturnCode
        */
        XPERSIF::ReturnCode addFeature(XPERSIFDT::VisionFeature visionfeature);

        /**
        * Remove a feature from the list of features to extract by the Vision Component.
        *
        * @param visionfeature in: The vision feature to remove
        * @return ReturnCode
        */
        XPERSIF::ReturnCode removeFeature(XPERSIFDT::VisionFeature visionfeature);

        /**
        * Called by TeleSimView to signal completion of the initialization process.
        *
        * @param sceneConfig in: The initial configuration of the scene
        * @param scale in: The scale needed to convert simulated values to SI units
        * @return ReturnCode
        */
        void teleSimViewUpdate(XPERSIFDT::NodeConfigList sceneConfig, double scale);
    };

    /**
    * The Vision component and its interface.
    */
    interface IVisionComponent extends
        IVision,
        IVisionSubject,
        IVisionFeatureVectorSubject {
    };
};
#endif _VISION_ICE
```

B.19 Motivation.ice

```
// File Motivation.ice
// Needs to implement Surprise::IObserverOfSurpriseVector, Curiosity::IObserverOfCuriosityVector
#ifndef _MOTIVATION_ICE
#define _MOTIVATION_ICE

#include "XPERSIF.ice"
```

```
#include "XPERSIFDT.ice"

module Motivation{
    /**
    * All Commands.
    */
    enum CommandType {
        monitorComponentState
    };

    /**
    * Observer of the Motivation component.
    */
    interface IObserverOfMotivation {
        void updateCalledByMotivation(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * Motivation as subject of other components which are observers of the
    * Motivation component
    */
    interface IMotivationSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfMotivation* observerMotivationRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfMotivation* observerMotivationRef);
    };

    /**
    * Observer of the MotivationVector
    */
    interface IObserverOfMotivationVector {
        void setMotivationVector(XPERSIFDT::MotivationVector motivationVector);
    };

    /**
    * MotivationVector as subject of other components which are observers of the
    * MotivationVector
    */
    interface IMotivationVectorSubject {
        XPERSIF::ReturnCode attachMotivationObserver(IObserverOfMotivationVector
                                                     *observerMotivationVectorRef);
        XPERSIF::ReturnCode detachMotivationObserver(IObserverOfMotivationVector
                                                     *observerMotivationVectorRef);
    };

    interface IMotivation extends XPERSIF::IOperation, XPERSIF::IOrganizationalOperation {
    };

    /**
    * The Motivation component and its interface.
    */
    interface IMotivationComponent extends
        IMotivation,
        IMotivationSubject,
        IMotivationVectorSubject    {
    };
```

```
};
#endif _MOTIVATION_ICE
```

B.20 Surprise.ice

```
// File Surprise.ice
// Needs to implement FeatureExtraction::IObserverOfFeatureVector
#ifndef _SURPRISE_ICE
#define _SURPRISE_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"

module Surprise{

    /**
    * All Commands.
    */
    //enum CommandType {
    //};

    /**
    * Observer of the Surprise component.
    */
    interface IObserverOfSurprise {
        void updateCalledBySurprise(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * Surprise as subject of other components which are observers of the
    * Surprise component
    */
    interface ISurpriseSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfSurprise *observerSurpriseRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfSurprise *observerSurpriseRef);
    };

    /**
    * Observer of the surprising predicates. Allows Surprise to set these predicates in an observer
    */
    interface IObserverOfSurpriseVector {
        XPERSIF::ReturnCode setSurpriseVector(XPERSIFDT::SurpriseVector surpriseVector);
    };

    /**
    * Surprise as subject of other components which are observers of the
    * SurpriseVector
    */
    interface ISurpriseVectorSubject {
        XPERSIF::ReturnCode attachSurpriseVectorObserver(IObserverOfSurpriseVector *observerSurpriseVectorRef
            );
        XPERSIF::ReturnCode detachSurpriseVectorObserver(IObserverOfSurpriseVector *observerSurpriseVectorRef
            );
    };
```

```
    interface ISurprise extends XPERSIF::IOperation {
    };

    /**
    * The Surprise component and its interface.
    */
    interface ISurpriseComponent extends
        ISurprise,
        ISurpriseSubject,
        ISurpriseVectorSubject
    {
    };
};
#endif _SURPRISE_ICE
```

B.21 Curiosity.ice

```
// File Curiosity.ice
// Needs to implement FeatureExtraction::IObserverOfFeatureVector
#ifndef _CURIOSITY_ICE
#define _CURIOSITY_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"

module Curiosity{
    /**
    * All Commands.
    */
    // enum CommandType {
    // };

    /**
    * Observer for the Curiosity component.
    */
    interface IObserverOfCuriosity {
        void updateCalledByCuriosity(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * Curiosity as subject of other components which are observers of the
    * Curiosity component
    */
    interface ICuriositySubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfCuriosity *observerCuriosityRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfCuriosity *observerCuriosityRef);
    };

    /**
    * Observer of the CuriosityVector.
    */
    interface IObserverOfCuriosityVector {
        XPERSIF::ReturnCode setCuriosityVector(XPERSIFDT::CuriosityVector curiosityVector);
    };
```

```
    /**
    * Curiosity as subject of other components which are observers of the
    * CuriosityVector
    */
    interface ICuriosityVectorSubject {
        XPERSIF::ReturnCode attachCuriosityVectorObserver(IObserverOfCuriosityVector
                                                *observerCuriosityVectorRef);
        XPERSIF::ReturnCode detachCuriosityVectorObserver(IObserverOfCuriosityVector
                                                *observerCuriosityVectorRef);
    };

    interface ICuriosity extends XPERSIF::IOperation {
    };

    /**
    * The Curiosity component and its interface.
    */
    interface ICuriosityComponent extends
        ICuriosity,
        ICuriositySubject,
        ICuriosityVectorSubject {
    };
};
#endif _CURIOSITY_ICE
```

B.22 MachineLearning.ice

```
// File MachineLearning.ice
// Needs to implement FeatureExtraction::IObserverOfFeatureVector
#ifndef _MACHINELEARNING_ICE
#define _MACHINELEARNING_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"

module MachineLearning{

    /**
    * All Commands.
    */
    enum CommandType {
        learn
    };

    /**
    * Observer of the MachineLearning component.
    */
    interface IObserverOfMachineLearning {
        void updateCalledByMachineLearning(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * MachineLearning as subject of other components which are observers of the
    * MachineLearning component
    */
```

```
    interface IMachineLearningSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfMachineLearning *observerMachineLearningRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfMachineLearning *observerMachineLearningRef);
    };

    interface IMachineLearning extends XPERSIF::IOperation {
        /**
        * Learn using the feature vectors collected in the experimentation phase.
        *
        * @Command
        * @precondition The experimentation phase finished.
        * @precondition FeaturesVectors have been collected.
        * @effect The Process the feature vectors and attemps to learn.
        *
        * @return CommandReturnCode
        */
        XPERSIF::CommandReturnCode learn();
    };

    /**
    * The MachineLearning component and its interface.
    */
    interface IMachineLearningComponent extends
        IMachineLearning,
        IMachineLearningSubject {
    };
};
#endif _MACHINELEARNING_ICE
```

B.23 Knowledgebase.ice

```
// File Knowledgebase.ice
#ifndef _KNOWLEDGEBASE_ICE
#define _KNOWLEDGEBASE_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"

module KnowledgeBase{
    /**
    * All Commands.
    */
    // enum CommandType {
    // };

    /**
    * Observer of the Knowledgebase component.
    */
    interface IObserverOfKnowledgebase {
        void updateCalledByKnowledgebase(XPERSIF::CommandTaskInfo commandTaskInfo);
    };

    /**
    * Knowledgebase as subject of other components which are observers of the
    * Knowledgebase component
```

```
    */
    interface IKnowledgebaseSubject {
        XPERSIF::ReturnCode attachObserver(IObserverOfKnowledgebase *observerKnowledgebaseRef);
        XPERSIF::ReturnCode detachObserver(IObserverOfKnowledgebase *observerKnowledgebaseRef);
    };

    interface IKnowledgebase extends XPERSIF::IOperation {
        /**
        * Update a specific hypothesis.
        *
        * @param hypothesis in: hypothesis to be updated
        * @return ReturnCode
        */
        XPERSIF::ReturnCode updateHypothesis(XPERSIFDT::Hypothesis hypothesis);

        /**
        * Add a new notion.
        *
        * @param notion in: notion to be added
        * @return ReturnCode
        */
        XPERSIF::ReturnCode addNotion(XPERSIFDT::Atom notion);

        /**
        * Delivers the information for the given model.
        *
        * @param modelID in: the modelID whose information is to be delivered
        * @param modelInfo out: the information about the given model
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getModelInfo(XPERSIFDT::ModelID modelID, out XPERSIFDT::Model modelInfo);
    };

    /**
    * The Knowledgebase component and its interface.
    */
    interface IKnowledgebaseComponent extends
        IKnowledgebase,
        IKnowledgebaseSubject {
    };
};
#endif _KNOWLEDGEBASE_ICE
```

B.24 XPERSimView.ice

```
// File XPERSimView.ice
#ifndef _XPERSIMVIEW_ICE
#define _XPERSIMVIEW_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"

module XPERSimView{
    /**
    * Observer for XPERSimView
```

```
    */
    interface IImageSimObserver {
        void setFrame(XPERSIFDT::ImageSim image);
    };

    /**
    * XPERSimView as subject of other components which are observers of the
    * XPERSimView component.
    */
    interface IImageSimSubject {
        XPERSIF::ReturnCode attachImageSimObserver(IImageSimObserver *imageSimRef);
        XPERSIF::ReturnCode detachImageSimObserver(IImageSimObserver *imageSimRef);
    };

    interface IXPERSimViewOperation {
        /**
        * Camera as subject of other components which are observers of the
        * Camera component (Simulated camera).
        */
        XPERSIF::ReturnCode fetchScene( out XPERSIFDT::NodeConfigList sceneConfig );
    };

    /**
    * The XPERSimView's interface.
    */
    interface IXPERSimView extends
        IXPERSimViewOperation,
        IImageSimSubject  {
    };
};
#endif _XPERSIMVIEW_ICE
```

B.25 TeleSimView.ice

```
// File TeleSimView.ice
// Needs to extends XPERSimView::IImageSimObserver
#ifndef _TELESIMVIEW_ICE
#define _TELESIMVIEW_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"
#include "XPERSimView.ice"

module TeleSimView{
    interface ITeleSimView extends XPERSimView::IImageSimObserver
    {
        /**
        * Delivers the last image received from the camera
        *
        * @param image out: The simulated image.
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getSimImage(out XPERSIFDT::ImageSim image);

        /**
```

```
        * Delivers the last image received from the camera
        *
        * @param image out: The time at which the image was taken.
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getImage(out double time);

        /**
        * Connects with TeleSimView
        *
        */
        void connect();

        /**
        * Connects with TeleSimView
        *
        */
        void disconnect();
    };
};
#endif _TELESIMVIEW_ICE
```

B.26 TeleView.ice

```
// File TeleView.ice
// Needs to extend IImageObserver
#ifndef _TELESIM_ICE
#define _TELESIM_ICE

#include "XPERSIF.ice"
#include "XPERSIFDT.ice"

module TeleView{

    interface ITeleView
    {
        /**
        * Delivers the last image received from the camera
        *
        * @param image out: The image.
        * @return ReturnCode
        */
        XPERSIF::ReturnCode getImage(out XPERSIFDT::Image image);
    };
};
#endif _TELESIM_ICE
```

Appendix C

Ice Grid node configuration

This appendix includes instructions on setting up an application on a server which serves as an Ice Grid node.

The file config.registry is used to configure the registry which provides the location service. Under normal circumstances, this registry is located on a server with a wellknown address. Other servers acting as grid nodes contain files, for example confif.grid, would contain the address of the registry.

Sample contents of config.registry

```
# Registry configuration
IceGrid.InstanceName=XPERO-IceGrid
# Registry properties
IceGrid.Registry.Client.Endpoints=tcp -h localhost -p 11002
IceGrid.Registry.Server.Endpoints=tcp -h localhost -p 11004
IceGrid.Registry.Internal.Endpoints=tcp -h localhost -p 11005
IceGrid.Registry.SessionManager.Endpoints=tcp -h localhost -p 11003
IceGrid.Registry.DynamicRegistration=1
IceGrid.Registry.Data=db/registry
IceGrid.Registry.Client.ThreadPool.Size=5
IceGrid.Registry.Client.ThreadPool.SizeMax=20
IceGrid.Registry.Client.ThreadPool.SizeWarn=18
IceGrid.Registry.PermissionsVerifier=XPERO-IceGrid/NullPermissionsVerifier
IceGrid.Registry.AdminPermissionsVerifier=XPERO-IceGrid/NullPermissionsVerifier
```

Sample contents of config.grid

```
IceGrid.InstanceName=XPERO-IceGrid
#
# The IceGrid locator proxy.
#
Ice.Default.Locator=XPERO-IceGrid/Locator:tcp -h localhost -p 11002
#
# IceGrid node configuration.
#
IceGrid.Node.Name=BRS
IceGrid.Node.Endpoints=tcp -p 12000
IceGrid.Node.Data=db/node
#IceGrid.Node.CollocateRegistry=1
#IceGrid.Node.Output=db
#IceGrid.Node.RedirectErrToOut=1
#
```

```
# Trace properties.
#
IceGrid.Node.Trace.Activator=1
IceGrid.Node.Trace.Patch=1
Ice.Trace.Network=2
Ice.Logger.Timestamp=1
```

The instructions below explain the configuration and initialization of the registry and a grid node so that they may run locally (on a single machine). The instructions are for the Windows operating system and assume that Ice 3.2.1 has already been installed on the server and that your application is located in the C:\XPERSIF\ directory.

- Open a console window (Windows Start menu: run: cmd) You will need three of these console windows.
- Change directory so that the prompt reads C:\XPERSIF
- Console 1:
  - Start the registry by typing icegridregistry –Ice.Config=config.registry
- Console 2:
  - Start the node by typing icegridnode –Ice.Config=config.grid
- Console 3:
  - Change your directory to C:\XPERSIF\XPERSIF
  - Start IceGrid Admin by typing icegridadmin –Ice.Config=config.grid
  - At the username and password prompt press any letter followed by enter.
  - Add the XPERSIF application by typing application add 'application.xml'
  - Type exit at the prompt.
  - Change your directory to C:\XPERSIF\YourApplicationsName
  - Start IceGrid Admin by typing icegridadmin –Ice.Config=config.grid
  - At the username and password prompt press any letter.
  - Add the XPERSIF application by typing application add 'application.xml'
  - Type exit at the prompt.

The steps under console 3 need only be done once.

Appendix D

Plan execution nodes

The nodes specified are:

- rotate and move forward (figure D.1). This allows the robot to move in straight lines to reach a pose instead of smoothly converging to a pose as is the case using the implemented controller (Appendix E).
- roam (figure D.2)
- approach - to view, to contact, to grip, to push (figure D.3)
- push and push from angle (figure D.4)
- grip object (figure D.5)
- carry object to pose (figure D.6)
- circumnavigate object (figure D.7)
- move with object in view (figure D.8). This results in a star like path for the robot circumnavigating an object.
- follow wall (figure D.9)

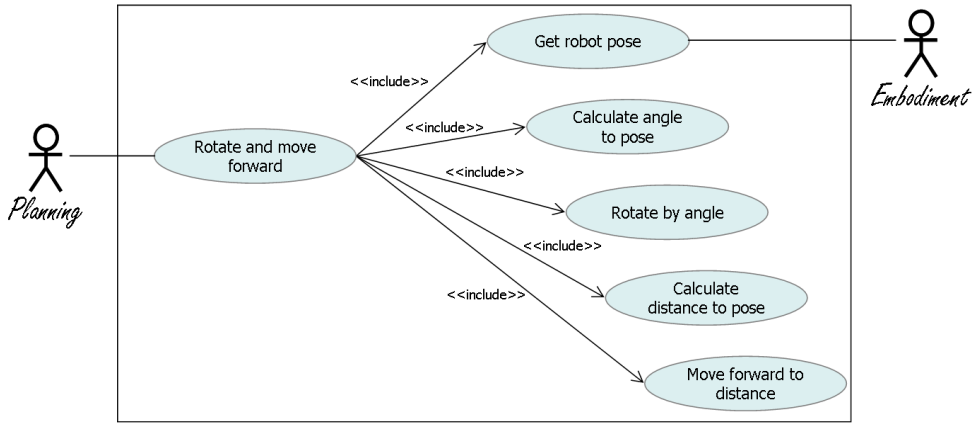

Figure D.1: The use case diagram for rotating and moving forward to a pose.

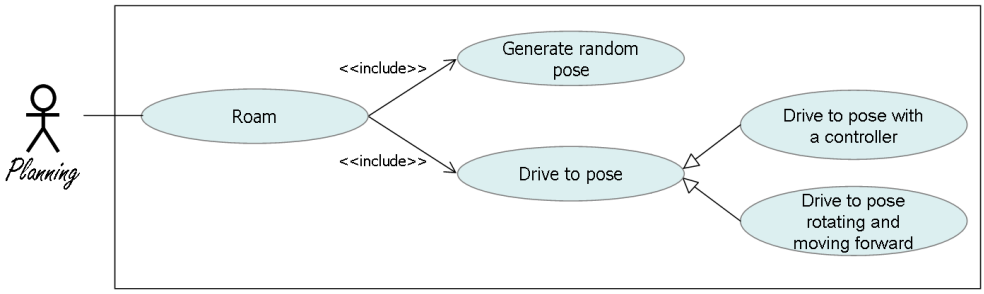

Figure D.2: The use case diagram for roaming.

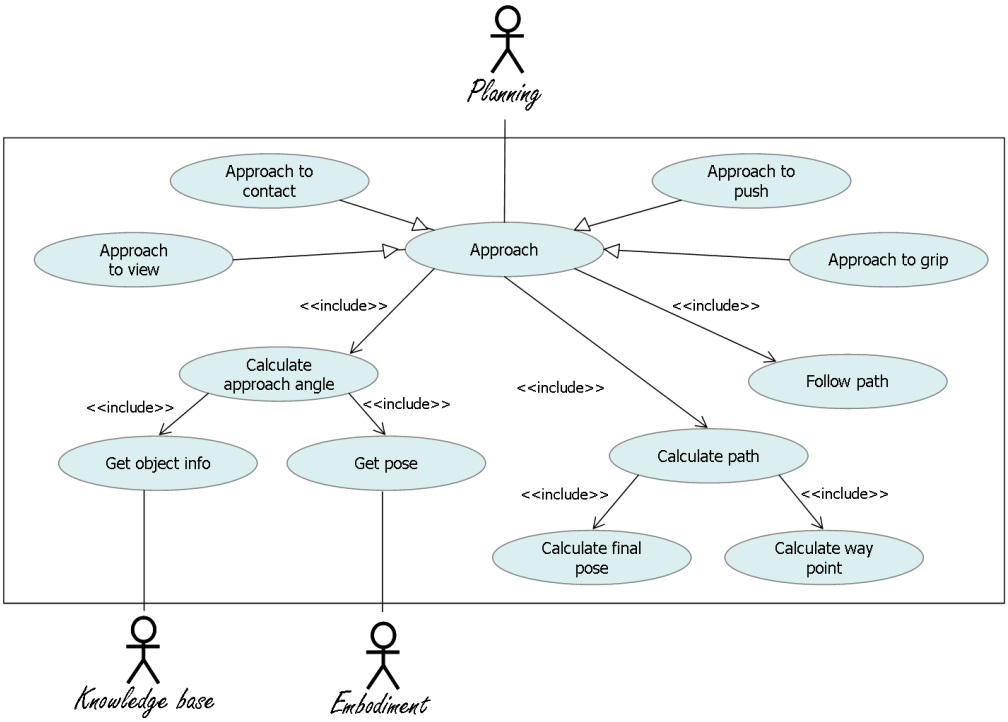

Figure D.3: The use case diagram for approaching an object such as a red box for example.

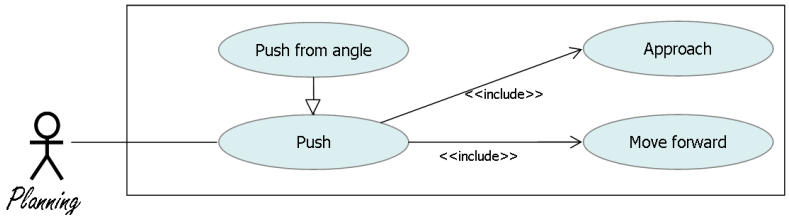

Figure D.4: The use case diagram for pushing an object.

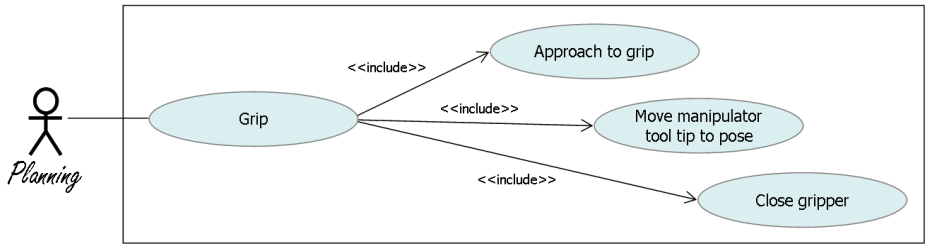

Figure D.5: The use case diagram for griping an object.

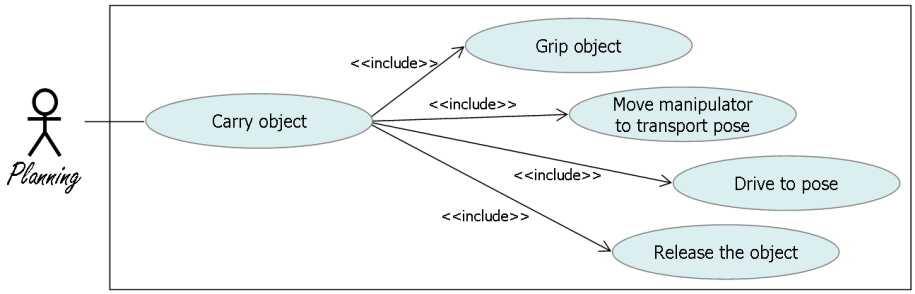

Figure D.6: The use case diagram for carrying an object to a pose.

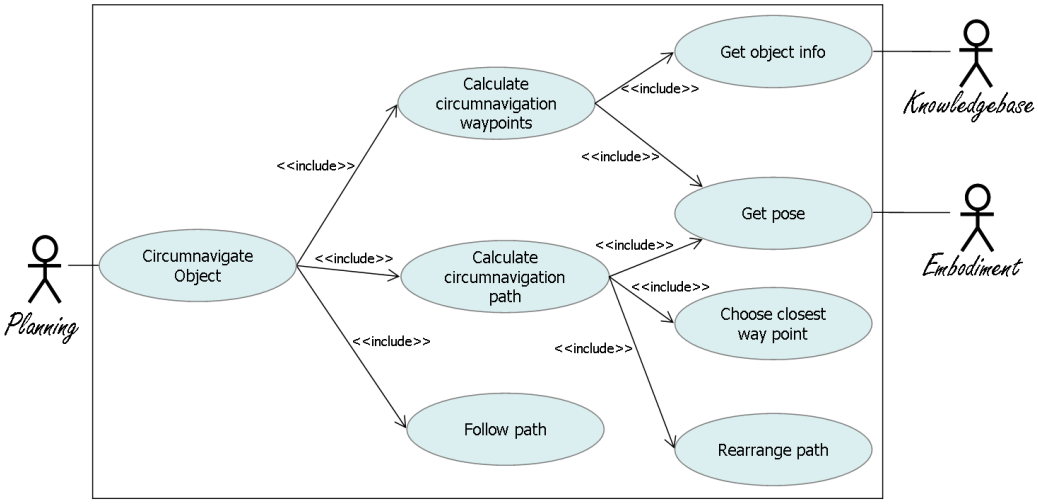

Figure D.7: The use case diagram for circumnavigating an object.

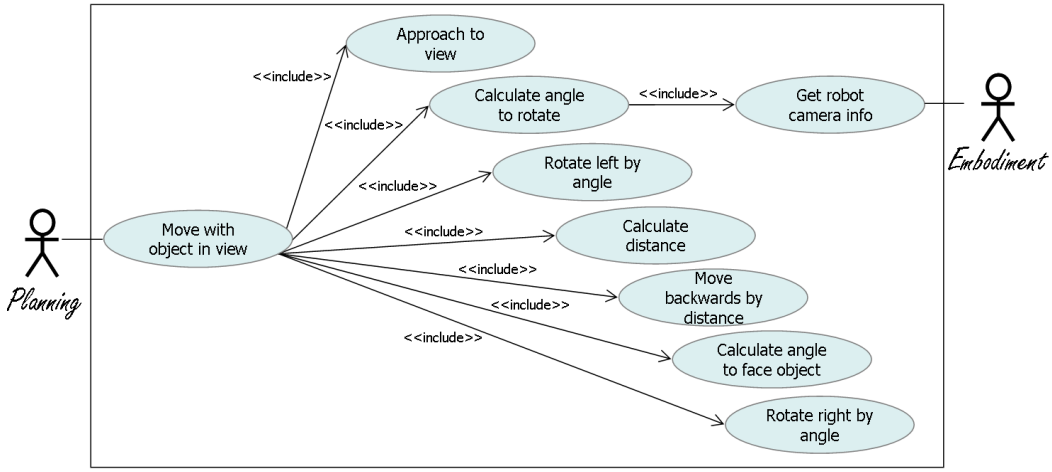

Figure D.8: The use case diagram for moving with an object in view an object.

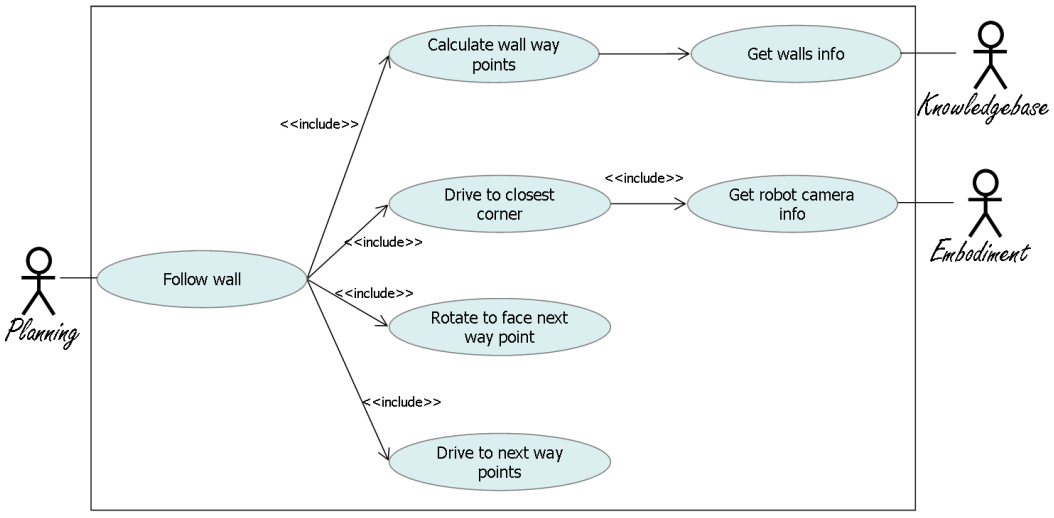

Figure D.9: The use case diagram for following a wall.

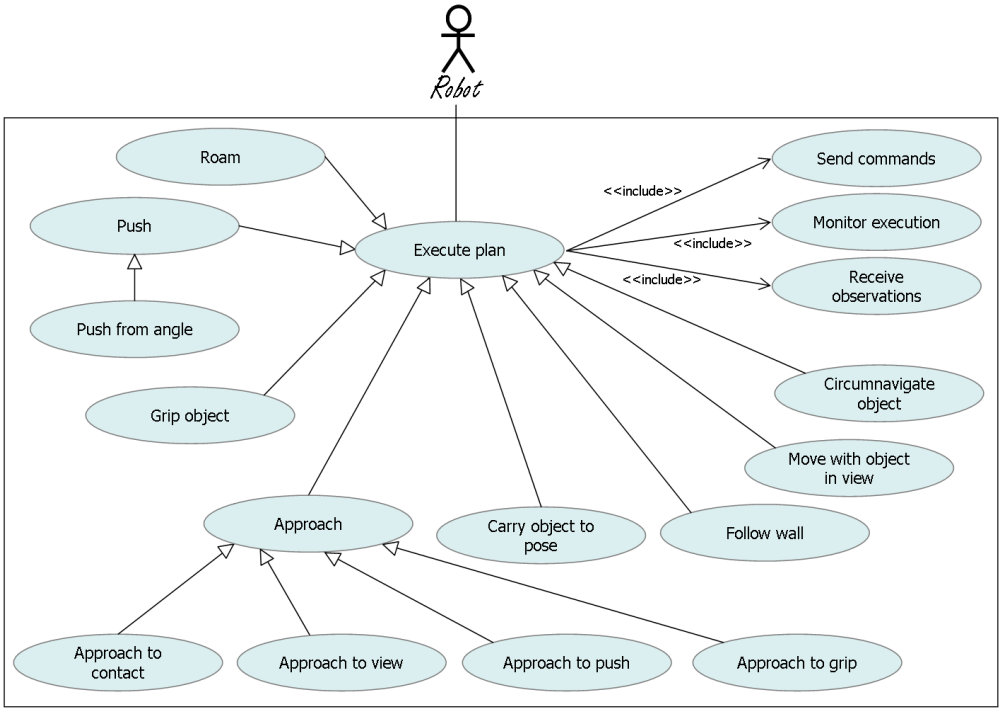

Figure D.10: The use case for plan execution.

Appendix E

The controller implementation

The details of the implemented controller is found in this appendix. Figure E.1 below shows the inverse kinematics problem which entails calculating the velocities needed to move a robot from one pose to another.

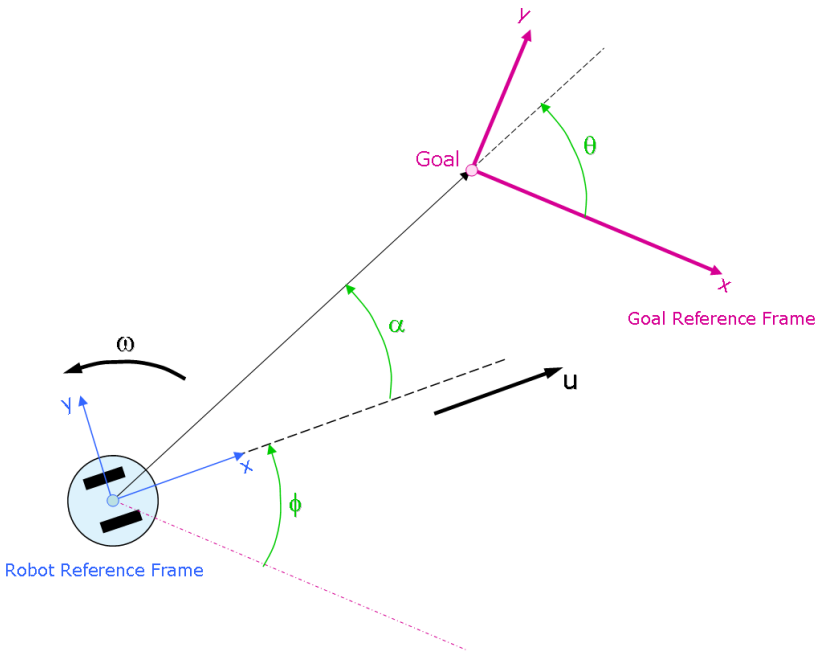

Figure E.1: The robot's position and orientation with respect to the target reference frame

In figure E.1, u and ω denote the linear and the angular velocity of the robot respectively. The shortest straight-line distance between the two poses is denoted by e and α is the angle between the robot's principal axis (x) and the distance vector e. θ is the orientation angle between the robot's principle axis in the goal's reference frame and that of the distance vector e.

The well-known kinematics equations for a robot (in the Cartesian coordinate frame) with pose x,y, ϕ are as follows:

$$\dot{x} = u\cos\phi$$
$$\dot{y} = u\sin\phi$$
$$\dot{\phi} = \omega$$

The controller which has been implemented uses the approach defined in [1] where the authors apply the Lyapunov theory to obtain effective closed-loop control laws for unicycle-like vehicles. The novelty of the authors' approach comes in the choice of the system of kinematic equations (system state equations) which facilitate designing appropriate closed-loop control laws for the vehicle maneuvering [1].

The choice of state-equations is transformed from the well-known kinematics equations above to the ones below by transforming the representation of the robot's pose to use polar coordinates:

$$\dot{e} = -u\cos\alpha$$
$$\dot{\alpha} = \omega + u\frac{\sin\alpha}{e}$$
$$\dot{\theta} = u\frac{\sin\alpha}{e}$$

By so doing, the authors do not break Brockett's law [7] while seemingly contradicting it by building a controller which asymptotically stabilizes by way of smooth and time-invariant feedback laws. They are able to do this as the regularity assumptions needed to apply Brockett's law do not hold for their choice of state equations [1]. The control laws which they present are summarized as follows:

$$u = (\gamma\cos\alpha)e \text{ where } \gamma > 0$$
$$\omega = k\alpha + \gamma\frac{\cos\alpha\sin\alpha}{\alpha}(\alpha + h\theta) \text{ where } k > 0$$

The parameters which need to be tuned are k, γ and h. The controller has been tested using the XPERSim simulator where the following values for k, γ and h provided good performance.

$k = 0.6$ $\gamma = 0.127$ $h = 2.0$

To illustrate the behavior of the controller, several trials for various goal poses where the final orientation of the robot changed were carried out. The results are shown in figure E.2.

The controller makes use of the observation component for localization. Dead-reckoning using the encoder sensors is also possible, however the accumulation of the error in the position of the robot makes the use of the observation component for localization much more effective.

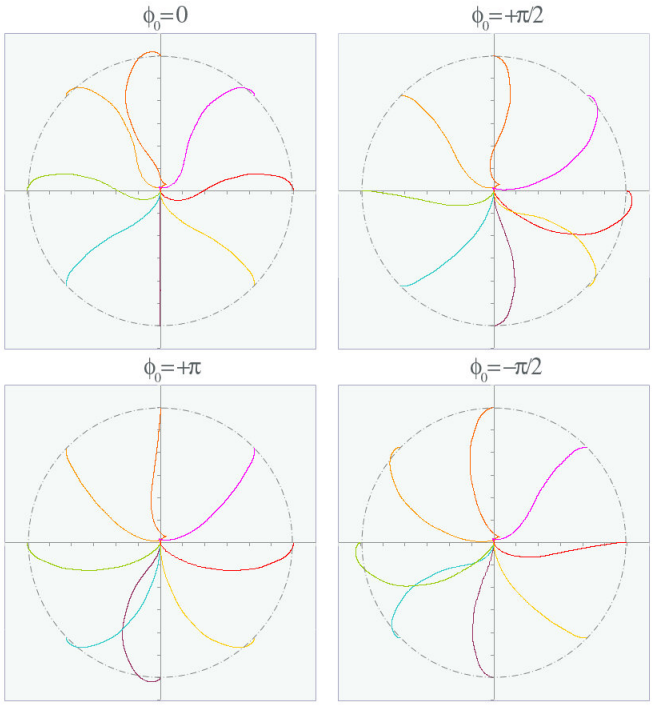

Figure E.2: Paths from the origin to poses located on the circumference of a circle with radius $0.3m$. From the top-left in clockwise direction the starting orientation Φ_0 is 0, $\Pi/2$, Π, $-\Pi/2$

It should be mentioned that the implemented controller allows behavior which is very similar to that of a car being driven in that the velocities are almost always positive, with very little movement in reverse (see Figure E.2. Moreover, as the control laws drive the angle α to zero, it is always the case that (at least) the final portion of the maneuver must be obtained by applying positive velocities. Should a position be given behind the robot, even in the case where it lies on a straight line from the current robot position and could easily be reached by applying equal negative velocities to the two wheels of a differentially driven robot, the controller will not in fact produce a maneuver with the negative velocities.

Since it is often the case that backward locomotion is in fact necessary (for example, to maintain an object of interest within the field of view, or to avoid pushing a nearby object away) it was necessary to implement two commands to reach poses (both absolute and relative) whilst driving in 'reverse'. These are the moveAbsInReverse and the moveRelInReverse commands. The same controller is used but the goal pose and the localization of the robot is changed by adding Π to the yaw values (ω). The controller then computes the necessary velocities for the maneuver, then, in the conversion to differential drive velocities, the values for the left and right wheels are switched. In such

a way, the same controller is used to arrive at a destination while facing the opposite direction.

As mentioned previously, a number of helper functions are implemented within the RelocationI class. A brief presentation of these helper functions follows.

The conversion from the velocity vector to left and right wheel speeds of a differential drive is performed in the function convertToDiffDriveVelocity using the following formulas:

$$V_l = V_x - \frac{V_\omega \times l}{2} \text{ and } V_r = V_x + \frac{V_\omega \times l}{2}$$

where V_l is the velocity of the left wheel, V_r is the velocity of the right wheel, V_x is the x component of the velocity vector, V_ω is the angular velocity and l is the distance between the wheels.

Similarly, the conversion from differential drive velocities is performed in the convertFromDiffDriveVelocity function using the following formulas:

$$V_x = \frac{V_l + V_r}{2} \; ; \; V_y = 0 \; ; \; V_\omega = \frac{V_l - V_r}{l}$$

Appendix F

Authorship

In accordance with German law, the authroship of the various parts of this thesis is presented here.

The introduction, results and conclusions were co-authored.
Chapters 3 and 5 were authored by Iman Awaad.
Chapters 2 and 4 were authored by Beatriz León.